Refine Search

Search Results -

Term	Documents
SELECT$	0
SELECT	321912
SELECTA	63
SELECTABBLE	1
SELECTABE	2
SELECTABEL	7
SELECTABILILTY	1
SELECTABILITY	969
SELECTABILITY/ACCESSIBILITY	1
SELECTABLE	74447
SELECTABLEATTENUATION	1
(SELECT$ AND L37).USPT.	1

There are more results than shown above. Click here to view the entire set.

Database:
US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L38

Refine Search | Recall Text | Clear | Interrupt

Search History

DATE: Saturday, March 20, 2004 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT; PLUR=YES; OP=ADJ		
L38	select$ and L37	1	L38

h e b b cg b e e ch

L37	l12 and information and version$1	1	L37
L36	l12 and information near10 version$1	0	L36
L35	l12 and information near4 version$1	0	L35
L34	l12 and version and architecture	1	L34
L33	l12 and appropriate	1	L33
L32	l12 and L31	0	L32
L31	compatibil$	79501	L31
L30	l12 and compatibil$	0	L30
L29	l12 and compatibl$	0	L29
L28	l12 and l26	1	L28
L27	l4 and L26	1	L27
L26	RAm	171991	L26
L25	L24	1	L25
L24	l4 and l15 and l18	1	L24
L23	l12 and L22	1	L23
L22	memor$ or storag$	879962	L22
L21	l12 and L20	0	L21
L20	ROM	131421	L20
L19	l12 and L18	0	L19
L18	volatile	163085	L18
L17	l12 and L15	0	L17
L16	l1 and L15	1	L16
L15	nonvolatile	28866	L15
L14	l12 and L13	1	L14
L13	designer	54740	L13
L12	5459854.pn.	1	L12
L11	l4 and L10	1	L11
L10	diskette	10568	L10
L9	l7 and L8	1	L9
L8	OP or operating system	91633	L8
L7	l5 and L6	1	L7
L6	hardware	181138	L6
L5	l2 and L4	1	L5
L4	5128995.pn.	1	L4
L3	l1 and L2	1	L3
L2	initial$	1066339	L2
L1	6430685.pn.	1	L1

END OF SEARCH HISTORY

h e b b cg b e e ch

Refine Search

Search Results -

Term	Documents
SELECT$	0
SELECT	321912
SELECTA	63
SELECTABBLE	1
SELECTABE	2
SELECTABEL	7
SELECTABILILTY	1
SELECTABILITY	969
SELECTABILITY/ACCESSIBILITY	1
SELECTABLE	74447
SELECTABLEATTENUATION	1
(SELECT$ AND L37).USPT.	1

There are more results than shown above. Click here to view the entire set.

Database:
US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L38

Refine Search | Recall Text | Clear | Interrupt

Search History

DATE: Saturday, March 20, 2004 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT; PLUR=YES; OP=ADJ		
L38	select$ and L37	1	L38

h e b b cg b e e ch

L37	l12 and information and version$1	1	L37
L36	l12 and information near10 version$1	0	L36
L35	l12 and information near4 version$1	0	L35
L34	l12 and version and architecture	1	L34
L33	l12 and appropriate	1	L33
L32	l12 and L31	0	L32
L31	compatibil$	79501	L31
L30	l12 and compatibil$	0	L30
L29	l12 and compatibl$	0	L29
L28	l12 and l26	1	L28
L27	l4 and L26	1	L27
L26	RAm	171991	L26
L25	L24	1	L25
L24	l4 and l15 and l18	1	L24
L23	l12 and L22	1	L23
L22	memor$ or storag$	879962	L22
L21	l12 and L20	0	L21
L20	ROM	131421	L20
L19	l12 and L18	0	L19
L18	volatile	163085	L18
L17	l12 and L15	0	L17
L16	l1 and L15	1	L16
L15	nonvolatile	28866	L15
L14	l12 and L13	1	L14
L13	designer	54740	L13
L12	5459854.pn.	1	L12
L11	l4 and L10	1	L11
L10	diskette	10568	L10
L9	l7 and L8	1	L9
L8	OP or operating system	91633	L8
L7	l5 and L6	1	L7
L6	hardware	181138	L6
L5	l2 and L4	1	L5
L4	5128995.pn.	1	L4
L3	l1 and L2	1	L3
L2	initial$	1066339	L2
L1	6430685.pn.	1	L1

END OF SEARCH HISTORY

h e b b cg b e e ch

Refine Search

Search Results -

Term	Documents
SELECT$	0
SELECT	321912
SELECTA	63
SELECTABBLE	1
SELECTABE	2
SELECTABEL	7
SELECTABILILTY	1
SELECTABILITY	969
SELECTABILITY/ACCESSIBILITY	1
SELECTABLE	74447
SELECTABLEATTENUATION	1
(SELECT$ AND L37).USPT.	1

There are more results than shown above. Click here to view the entire set.

Database:
US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L38

Refine Search

Recall Text Clear Interrupt

Search History

DATE: Saturday, March 20, 2004 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT; PLUR=YES; OP=ADJ		
L38	select$ and L37	1	L38

h e b b cg b e e ch

L37	l12 and information and version$1	1	L37
L36	l12 and information near10 version$1	0	L36
L35	l12 and information near4 version$1	0	L35
L34	l12 and version and architecture	1	L34
L33	l12 and appropriate	1	L33
L32	l12 and L31	0	L32
L31	compatibil$	79501	L31
L30	l12 and compatibil$	0	L30
L29	l12 and compatibl$	0	L29
L28	l12 and l26	1	L28
L27	l4 and L26	1	L27
L26	RAm	171991	L26
L25	L24	1	L25
L24	l4 and l15 and l18	1	L24
L23	l12 and L22	1	L23
L22	memor$ or storag$	879962	L22
L21	l12 and L20	0	L21
L20	ROM	131421	L20
L19	l12 and L18	0	L19
L18	volatile	163085	L18
L17	l12 and L15	0	L17
L16	l1 and L15	1	L16
L15	nonvolatile	28866	L15
L14	l12 and L13	1	L14
L13	designer	54740	L13
L12	5459854.pn.	1	L12
L11	l4 and L10	1	L11
L10	diskette	10568	L10
L9	l7 and L8	1	L9
L8	OP or operating system	91633	L8
L7	l5 and L6	1	L7
L6	hardware	181138	L6
L5	l2 and L4	1	L5
L4	5128995.pn.	1	L4
L3	l1 and L2	1	L3
L2	initial$	1066339	L2
L1	6430685.pn.	1	L1

END OF SEARCH HISTORY

h e b b cg b e e ch

Refine Search

Search Results -

Term	Documents
(50 AND 43).USPT.	1
(L43 AND L50).USPT.	1

Database:
- US Pre-Grant Publication Full-Text Database
- US Patents Full-Text Database
- US OCR Full-Text Database
- EPO Abstracts Database
- JPO Abstracts Database
- Derwent World Patents Index
- IBM Technical Disclosure Bulletins

Search: L51

Refine Search

Recall Text | Clear | Interrupt

Search History

DATE: Saturday, March 20, 2004 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L51	l43 and L50	1	L51
L50	l24 and l26 and L49	1	L50
L49	6496871.pn.	1	L49
L48	l44 and constant pool	2	L48
L47	l44 and L46	0	L47
L46	class representation	97	L46
L45	l15 and L44	0	L45
L44	l42 and L43	3	L44
L43	link$ and runtime	3283	L43
L42	l6 and L41	6	L42
L41	l10 and l5 and L40	6	L41
L40	l26 and L39 and l24	10	L40
L39	class pointer	200	L39

h e b b cg b e e ch

L38	l32 and L37	0	L38
L37	internal representation near3 call	7	L37
L36	reference cell and l15	0	L36
L35	L24 and l34	1	L35
L34	l10 and L33	4	L34
L33	l5 and L32	6	L33
L32	reference cell	3786	L32
L31	l3 and L30	1	L31
L30	invok$ near3 method	3477	L30
L29	l3 and L28	1	L29
L28	method near3 name	3844	L28
L27	l3 and L26	0	L27
L26	method name	792	L26
L25	l3 and L24	1	L25
L24	signature	25048	L24
L23	signiture	14	L23
L22	l10 and L21	1	L22
L21	L20 and (l14 or l15)	1	L21
L20	L19 and l6	1	L20
L19	l3 and l5	1	L19
L18	l2 and l5	1	L18
L17	l2 and l5 and l15	0	L17
L16	l2 and l5 and l10 and l15	0	L16
L15	internal class	83	L15
L14	inner class	30	L14
L13	pointer$1	64908	L13
L12	method$1	2070068	L12
L11	mtthod$1	6	L11
L10	Virtual	61408	L10
L9	L8 and method	1	L9
L8	l5 and l2	1	L8
L7	l5 and L6 and l2	0	L7
L6	method and pointer	51885	L6
L5	class and java	3910	L5
L4	6081665.pn.	1	L4
L3	6643711.pn.	1	L3
L2	6434694.pn.	1	L2
L1	6434694.pn.	1	L1

END OF SEARCH HISTORY

h e b b cg b e e ch

L8: Entry 11 of 11 File: USPT Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A
TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Abstract Text (1):
The invention is a method for use in executing portable virtual machine computer programs under real-time constraints. The invention includes a method for implementing a single abstract virtual machine execution stack with multiple independent stacks in order to improve the efficiency of distinguishing memory pointers from non-pointers. Further, the invention includes a method for rewriting certain of the virtual machine instructions into a new instruction set that more efficiently manipulates the multiple stacks. Additionally, using the multiple-stack technique to identify pointers on the run-time stack, the invention includes a method for performing efficient defragmenting real-time garbage collection using a mostly stationary technique. The invention also includes a method for efficiently mixing a combination of byte-code, native, and JIT-translated methods in the implementation of a particular task, where byte-code methods are represented in the instruction set of the virtual machine, native methods are written in a language like C and represented by native machine code, and JIT-translated methods result from automatic translation of byte-code methods into the native machine code of the host machine. Also included in the invention is a method to implement a real-time task dispatcher that supports arbitrary numbers of real-time task priorities given an underlying real-time operating system that supports at least three task priority levels. Finally, the invention includes a method to analyze and preconfigure virtual memory programs so that they can be stored in ROM memory prior to program.

Brief Summary Text (4):
Java (a trademark of Sun Microsystems, Inc.) is an object-oriented programming language with syntax derived from C and C++. However, Java's designers chose not to pursue full compatibility with C and C++ because they preferred to eliminate from these languages what they considered to be troublesome features. In particular, Java does not support enumerated constants, pointer arithmetic, traditional functions, structures and unions, multiple inheritance, goto statements, operator overloading, and preprocessor directives. In their place, Java requires all constant identifiers, functions (methods), and structures to be encapsulated within

Brief Summary Text (5):
class (object) declarations. The purpose of this requirement is to reduce conflicts in the global name space. Java provides standardized support for multiple threads (lightweight tasks) and automatic garbage collection of dynamically-allocated memory. Furthermore, Java fully specifies the behavior of every operator on every type, unlike C and C++ which leave many behaviors to be implementation dependent. These changes were designed to improve software scalability, reduce software development and maintenance costs, and to achieve full portability of Java software. Anecdotal evidence suggests that many former C and C++ programmers have welcomed these language improvements.

h e b b g e e e f c e f h e ge

Brief Summary Text (6):
One distinguishing characteristic of Java is its execution model. Java programs are first translated into a fully portable standard byte code representation. The byte code is then available for execution on any Java virtual machine. A Java virtual machine is simply any software system that is capable of understanding and executing the standard Java byte code representation. Java virtual machine support is currently available for AIX, Apple Macintosh, HPUX, Linux, Microsoft NT, Microsoft Windows 3.1, Microsoft Windows 95, MVS, Silicon Graphics IRIX, and Sun Solaris. Ports to other environments are currently in progress. To prevent viruses from being introduced into a computer by a foreign Java byte-code program, the Java virtual machine includes a Java byte code analyzer that verifies the byte code does not contain requests that would compromise the local system. By convention, this byte code analyzer is applied to every Java program before it is executed. Byte code analysis is combined with optional run-time restrictions on access to the local file system for even greater security. Current Java implementations use interpreters to execute the byte codes but future high-performance Java systems will have the capability of translating byte codes to native machine code on the fly. In theory, this will allow Java programs to run approximately at the same speed as C++.

Brief Summary Text (7):
Within Sun, development of Java began in April of 1991. Initially, Java was intended to be an implementation language for personal digital assistants. Subsequently, the development effort was retargeted to the needs of set-top boxes, CD-ROM software, and ultimately the World-Wide Web. Most of Java's recent media attention has focused on its use as a medium for portable distribution of software over the Internet. However, both within and outside of Sun, it is well understood that Java is much more than simply a language for adding animations to Web pages. In many embedded real-time applications, for example, the Java byte codes might be represented in system ROMs or might even be pre-translated into native machine code.

Brief Summary Text (8):
Many of the more ambitious "industrial-strength" sorts of applications that Java promises to enable on the Internet have associated real-time constraints. These applications include video conferencing integrated with distributed white boards, virtual reality, voice processing, full-motion video and real-time audio for instruction and entertainment, and distributed video games. More importantly, the next generation Web client will have even more real-time requirements. Future set-top devices will connect home televisions to the Web by way of cable TV networks. Besides all of the capabilities just mentioned, these systems will also support fully interactive television applications.

Brief Summary Text (9):
Java offers important software engineering benefits over C and C++, two of the more popular languages for current implementation of embedded real-time systems. If Java could be extended in ways that would allow it to support the cost-effective creation of portable, reliable real-time applications, the benefits of this programming language would be realized by a much larger audience than just the people who are implementing real-time Web applications. All developers of embedded real-time software could benefit. Some of the near-term applications for which a real-time dialect of Java would be especially well suited include personal digital assistants, real-time digital diagnosis (medical instrumentation, automotive repair, electronics equipment), robotics, weather monitoring and forecasting, emergency and service vehicle dispatch systems, in-vehicle navigation systems, home and business security systems, military surveillance, radar and sonar analysis, air traffic control, and various telephone and Internet packet switching applications.

Brief Summary Text (10):
This invention relates generally to computer programming methods pertaining to

real-time applications and more specifically to programming language implementation methods which enable development of real-time software that can run on computer systems of different designs. PERC (a trademark of NewMonics Inc.) is a dialect of the Java programming language designed to address the special needs of developers of real-time software.

Brief Summary Text (11):
PERC has much to offer developers of embedded real-time systems. High-level abstractions and availability of reusable software components shorten the time-to-market for innovative products. Its virtual machine execution model eliminates the need for complicated cross-compiler development systems, multiple platform version maintenance, and extensive rewriting and retesting each time the software is ported to a new host processor. It is important to recognize that the embedded computing market is quite large. Industry observers have predicted that by the year 2010, there will be ten times as many software programmers writing embedded systems applications as there will be working on software for general purpose computers.

Brief Summary Text (12):
Unlike many existing real-time systems, most of the applications for which PERC is intended are highly dynamic. New real-time workloads arrive continually and must be integrated into the existing workload. This requires dynamic management of memory and on-the-fly schedulability analysis. Price and performance issues are very important, making certain traditional real-time methodologies cost prohibitive. An additional complication is that an application developer is not able to test the software in each environment in which it is expected to run. The same Java byte-code application would have to run within the same real-time constraints on a 50 MHz 486 and on a 300 MHz Digital Alpha. Furthermore, each execution environment is likely to have a different mix of competing applications with which this code must contend for CPU and memory resources. Finally, every Java byte-code program is supposed to run on every Java virtual machine, even a virtual machine that is running as one of many tasks executing on a time-sharing host. Clearly, time-shared virtual machines are not able to offer the same real-time predictability as a specially designed PERC virtual machine embedded within a dedicated microprocessor environment. Nevertheless, such systems are able to provide soft-real-time response.

Brief Summary Text (15):
Byte code is a term of art that describes a method of encoding instructions (for interpretation by a virtual machine) as 8-bit numbers, each pattern of 8 bits representing a different instruction.

Brief Summary Text (26):
Java, a trademark of Sun Microsystems, Inc., is an object-oriented programming language with syntax derived from C and C++, which provides automatic garbage collection and multi-threading support as part of the standard language definition.

Brief Summary Text (27):
JIT, as the term is used in this invention disclosure, is an acronym standing for "just in time." The term is used to describe a system for translating Java byte codes to native machine language on the fly, just-in-time for its execution. We consider any translation of byte code to machine language which is carried out by the virtual machine to be a form of JIT compilation.

Brief Summary Text (30):
Native Method, as this term is used in relation to the Java and PERC programming languages, describes a method that is implemented in C (or some other low-level language) rather than in the high-level Java or PERC language in which the majority of methods are implemented.

Brief Summary Text (31):
PERC, a trademark of NewMonics Inc., is an object-oriented programming language with similarities to Java, which has been designed to address the specific needs of developers of real-time and embedded software.

Brief Summary Text (37):
RTVMM, as the term is used in this invention disclosure, is an acronym standing for Real-Time Virtual Machine Method. This acronym represents the invention disclosed by this document.

Brief Summary Text (43):
Thread is a term of art describing a computer program that executes with an independent flow of execution. Java is a threaded language, meaning that multiple flows of execution may be active concurrently. All threads share access to the same global memory pool. (In other programming environments, threads are known as tasks.)

Brief Summary Text (44):
Virtual Machine is a term of art that describes a software system that is capable of interpreting the instructions encoded as numbers according to a particular agreed upon convention.

Brief Summary Text (47):
The invention is a real-time virtual machine method (RTVMM) for use in implementing portable real-time systems. The RTVMM provides efficient support for execution of portable byte-code representations of computer programs, including support for accurate defragmenting real-time garbage collection. Efficiency is measured both in terms of memory utilization, CPU time, and programmer productivity. Programmer productivity is enhanced through reduction of the human effort required to make the RTVMM available in multiple execution environments.

Brief Summary Text (49):
1. Extensions to the standard Java byte code instruction set to enable efficient run-time isolation of pointer variables from non-pointer variables. The extended byte codes are described as the PERC instruction set.

Brief Summary Text (50):
2. A mechanism to translate traditional Java byte codes into the extended PERC byte codes at run-time, as new Java byte codes are loaded into the virtual machine's execution environment.

Brief Summary Text (52):
of the PERC instruction set. The Java run-time stack is replaced by two stacks, one for non-pointer and the other for pointer data. Further, the data structures enable efficient interaction between native methods, Java methods represented by byte code, and Java methods translated by a JIT compiler to native machine language. Performance tradeoffs are biased to give favorable treatment to execution of JIT-translated methods.

Brief Summary Text (56):
7. A mechanism for translating traditional Java byte codes into the extended PERC byte codes prior to run-time, in order to reduce run-time overhead and simplify system organization.

Drawing Description Text (6):
FIG. 5 illustrates the appearance of the pointer and non-pointer stack activation frames immediately before calling and immediately following entry into the body of a Java method. The stacks are assumed to grow downward. In preparation for the call, arguments are pushed onto the stack. Within the called method, the frame pointer (fp) is adjusted to point at the memory immediately above the first pushed

h e b b g e e e f c e f h e ge

Ate Emblant pool for the first time

L48: Entry 1 of 2 File: USPT Sep 3, 2002

DOCUMENT-IDENTIFIER: US 6446084 B1
TITLE: Optimizing symbol table lookups in platform-independent virtual machines

Abstract Text (1):
One embodiment of the present invention provides a method for increasing performance of code executing on a platform-independent virtual machine. The method operates by receiving a request to resolve an entry in a symbol table at run-time, wherein resolving the entry requires multiple lookups into the symbol table. It next determines if the entry has previously been resolved. If so, the system returns a direct pointer to a runtime structure associated with the entry, which was returned during a previous resolution of the entry. If not, the system resolves the entry through multiple lookups into the symbol table to produce a direct pointer to the runtime structure, and replaces the entry with the direct pointer. In a variation on the above embodiment, the symbol table assumes the form of a constant pool within an object-oriented class file defined within the JAVA programming language. The present invention speeds up constant pool resolution substantially without requiring a significant amount of additional space. Therefore, the present invention is especially valuable for embedded JAVA systems or other applications that have strict size limitations.

Brief Summary Text (5):
The present invention relates to platform-independent virtual machines for computer systems. More specifically, the present invention relates to a space-efficient mechanism for improving the speed of lookups of symbolic information related to functions, variables and object-oriented classes on platform-independent virtual machines.

Brief Summary Text (7):
The recent proliferation of computer networks such as the Internet has lead to the development of computer languages, such as the JAVA.TM. programming language distributed by Sun Microsystems, Inc. of Palo Alto, Calif. One important feature of the JAVA programming language is the way in which it allows components of a program to be loaded dynamically at runtime. This is accomplished by storing the components of the program as "class files" that can be easily transferred over a network such as the Internet to remote computer nodes. On the remote nodes, a platform-independent virtual machine can execute the program components stored within the class files.

Brief Summary Text (8):
An essential part of a JAVA classfile is the "constant pool," which is a type of symbol table that stores symbolic information for the associated JAVA class. This allows the JAVA virtual machine (JVM) to dynamically resolve references to functions, variables and other classes at runtime.

Brief Summary Text (9):
Sun, the Sun logo, Sun Microsystems, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Brief Summary Text (10):
Unfortunately, constant pool resolution is an expensive, time-consuming operation

h e b b g e e e f c e g g e ge

that occurs very frequently when JAVA programs are run. Most JVMs utilize certain techniques to reduce the need for such constant pool lookups. One commonly used technique is to introduce so-called "quick" bytecodes. This is done by dynamically replacing those JAVA bytecodes that necessitate a constant pool lookup with other "quick" bytecodes that do not require a constant pool lookup, but instead contain a direct pointer to the desired runtime structure. This replacement is performed dynamically when the original bytecode is executed and the associated constant pool references are resolved for the first time. The second time the virtual machine encounters the same code it no longer has to perform the constant pool lookups.

Brief Summary Text (11):
Even simple optimizations such as quick bytecodes usually result in 2-4 times faster execution time. However, these techniques make the virtual machine larger. The code needed to implement quick bytecodes typically increases the size of the virtual machine by at least 5-10 kilobytes, often substantially more if special cache areas are allocated to store the original bytecode sequences. Consequently, these techniques may not be practical for those applications where it is important to have the smallest possible JVM.

Brief Summary Text (12):
Another solution is to provide JVMs with a Just-In-Time (JIT) compiler. This type of system can avoid constant pool lookups by dynamically compiling JAVA bytecodes and the necessary constant pool information into machine code. However, JIT compilers typically require hundreds of kilobytes of additional memory space at the minimum. Consequently, they are completely unsuitable for embedded systems where the virtual machine has to be as small as possible.

Brief Summary Text (13):
What is needed is a space-efficient mechanism for improving the performance of the constant pool lookup process for platform-independent virtual machines.

Brief Summary Text (15):
One embodiment of the present invention provides a method for increasing performance of code executing on a platform-independent virtual machine. The method operates by receiving a request to resolve an entry in a symbol table at run-time, wherein resolving the entry requires multiple lookups into the symbol table. It next determines if the entry has previously been resolved. If so, the system returns a direct pointer to a runtime structure associated with the entry, which was returned during a previous resolution of the entry. If not, the system resolves the entry through multiple lookups into the symbol table to produce a direct pointer to the runtime structure, and replaces the entry with the direct pointer. In a variation on the above embodiment, the symbol table assumes the form of a constant pool within an object-oriented class file defined within the JAVA programming language. The present invention speeds up constant pool resolution substantially without requiring a significant amount of additional space. Therefore, the present invention is especially valuable for embedded JAVA systems or other applications that have strict size limitations.

Drawing Description Text (2):
FIG. 1 illustrates a computer system that loads a class file onto a compact computing device in accordance with an embodiment of the present invention.

Drawing Description Text (3):
FIG. 2 illustrates the structure of a class file and associated runtime structures in accordance with an embodiment of the present invention.

Drawing Description Text (5):
FIG. 4 is a flow chart illustrating a modified constant pool resolution process in accordance with an embodiment of the present invention.

Drawing Description Text (7):
FIG. 6A presents an example of a piece of code defines an object-oriented programming class in accordance with an embodiment of the present invention.

Detailed Description Text (4):
In the following disclosure and preceding discussion, many of the structures are described in terms of the JAVA programming language and supporting utilities. However, the present invention is not limited to implementations involving JAVA programming language. The present invention applies to any programming environment that supports platform-independent virtual machines. Hence, any mention of a JAVA programming language feature or associated utility is meant to apply to analogous structures in other systems that support program execution on platform-independent virtual machines.

Detailed Description Text (6):
FIG. 1 illustrates one embodiment of a computer system 106, which loads a class file onto a compact computing device 110 in accordance with an embodiment of the present invention. In FIG. 1, computer system 106 may be any type of computer system capable of executing an application program. This includes, but is not limited to, a personal computer, a workstation, a mainframe computer, and even a device controller. Computer system 106 contains JAVA development unit 108, which includes programming tools for developing JAVA applications. A user 102 operates computer system 106 and views the output of computer system 106 through display 104.

Detailed Description Text (7):
Computer system 106 is coupled to compact computing device 110 through a communication link 112. Compact computing device 110 may be any type of computing device including a limited amount of storage for code and data.

Detailed Description Text (9):
Communication link 112 may include any type of permanent or temporary communication channel that can be used to transfer data from computer system 106 to compact computing device 110. This may include, but is not limited to, a computer network such as an Ethernet, a wireless communication network or a telephone line. In some embodiments of the present invention, compact computing device 110 is designed to operate while it is disconnected from communication link 112.

Detailed Description Text (10):
Compact computing device 110 includes database 114, for storing code and data, as well as a platform-independent virtual machine 116 for processing platform-independent programs received across communication link 112.

Detailed Description Text (11):
During operation, class file 118 is created within JAVA development unit 108. Class file 118 contains components of a platform-independent program to be executed in compact computing device 110. For example, class file 118 may include methods and fields associated with an object-oriented class. Class file 118 additionally includes constant pool 206 as is described in more detail below. Next, class file 118 is transferred from JAVA development unit 108 through communication link 112, and into database 114 within compact computing device 110. Finally, virtual machine 116 executes a program that accesses components within class file 118. These accesses cause time-consuming constant pool resolution operations, which are optimized by this invention.

Detailed Description Text (12):
Class File and Run Time Structures

Detailed Description Text (13):
FIG. 2 illustrates the structure of class file 118 and associated runtime

structures in accordance with an embodiment of the present invention. FIG. 2 includes class file 118, method table 216, field table 220 and list of classes 222. Class file 118 includes a number of different types of information related to a particular class, including class file identification information 204, constant pool 206, general class information 208, field information 210 and method information 212. Class identification information 204 contains information that identifies the particular class. Constant pool 206 includes a number of entries for storing symbolic information for the particular class. General class information 208 includes information that identifies the superclass to which the particular class belongs. Field information 210 includes information relating to the various variables and data structures associated with the particular class. Method information 212 includes the actual bytecodes to implement the methods defined for the particular class.

Detailed Description Text (14):
Method table 216 includes pointers to the actual bytecodes that implement the methods defined within the particular class. This includes bytecode 218, which includes a string of bytes to be executed by virtual machine 116 in FIG. 1.

Detailed Description Text (15):
Field table 220 includes the values of fields associated with the particular class. The entries in field table 220 are typically stored as data values. However, they may additionally include pointers to data values.

Detailed Description Text (16):
Finally, list of classes 222 includes a list of classes that have already been loaded into virtual machine 116 in FIG. 1. Once these classes are loaded, components within the classes can be executed by virtual machine 116.

Detailed Description Text (17):
The structures illustrated in FIG. 2 operate as follows. During execution of a bytecode in virtual machine 116, references are generated to various methods, interface methods, fields or classes. These references are resolved through accesses to constant pool 206. When a constant pool entry is resolved for the first time, the data fields of the entry are replaced with a pointer to the class, method, field or interface method structure that was returned as a result of the resolution. Additionally, the corresponding tag in the constant pool entry is modified to indicate that the constant pool entry has been resolved. The next time virtual machine 116 accesses the same constant pool entry, it simply reads the value stored in the data fields of the entry instead of performing a full constant pool lookup. Note that this requires to additional data storage space. Also note that modifying the tag field adds no additional storage overhead to the constant pool.

Detailed Description Text (18):
Only a few tens of bytes of code are needed for implementing the extra caching instructions. In terms of execution speed the solution requires only a few extra logical AND and OR operations for checking the cache status. This overhead is easily offset by a dramatic speed-up in constant pool access.

Detailed Description Text (19):
The proposed technique adds some extra requirements for the garbage collector of the virtual machine. In particular, the garbage collector must be informed of the possible pointers in the constant pool.

Detailed Description Text (21):
FIG. 3 is a flow chart illustrating how bytecodes are executed in accordance with an embodiment of the present invention. The execution engine within the platform-independent virtual machine 116 first retrieves a byte code from the current instruction pointer (IP) (state 302). After the byte code is retrieved, the

instruction pointer is incremented to point to a subsequent byte code (state 304). Next, the retrieved byte code is invoked. This may generate a lookup in constant pool 206. If so, a constant pool resolution is performed, which may involve multiple lookups in constant pool 206 (state 306). Performing these multiple lookups can be very time-consuming. After the byte code is invoked, the system returns to state 302 to retrieve another byte code. The above process is repeated for subsequent byte codes executed by virtual machine 116.

Detailed Description Text (22):
FIG. 4 is a flow chart illustrating a modified constant pool resolution process in accordance with an embodiment of the present invention. The system first determines if the accessed entry in the constant pool has already been resolved (state 402). If so, the system returns a direct pointer to the structure specified by the entry, which was returned during a previous resolution of the entry (state 403). The constant pool resolution is complete. Otherwise, the system takes one of several courses of action depending upon what is contained in the entry. If the constant pool entry corresponds to a variable, the constant pool entry contains either the value of the variable or a direct pointer to the variable. Hence, no optimization is required to reduce the number of constant pool lookups. (This case is not shown.) If the constant pool entry requires multiple constant pool lookups, the system takes one of several actions depending upon if the constant pool entry corresponds to a method, an interface method, a field or a class (state 404).

Detailed Description Text (23):
If the entry corresponds to a method (or an interface method), the system first determines the class associated with the method by retrieving an index for the class (state 406) and then calling a class resolution subroutine to resolve the class (state 408). Next, the system retrieves an index for the name and type of the method (state 410), and uses this index to call a name and type subroutine to resolve the name and type (state 412). The system uses the class pointer, the method name and the type information (signature) to lookup a method pointer in method table 216 from FIG. 2 (state 414). Finally, the system returns this method pointer (state 416). An example code listing for the method resolution process appears in Table 1.

Detailed Description Text (24):
If the entry corresponds to a field, the system first determines the class associated with the field by retrieving an index for the class (state 418). The system uses this index to call the class resolution subroutine to resolve the class (state 420). Next, the system retrieves an index for the name and type of the field (state 422), and uses this index to call a name and type subroutine to resolve the name and type (state 424). The system uses the class pointer, the field name and the type information (signature) to lookup a field pointer in field table 220 from FIG. 2 (state 428). Finally, the system then returns the field pointer (state 430). The above field resolution process is essentially the same as the method resolution process, which appears in Table 1.

Detailed Description Text (25):
If the entry corresponds to a class, the system first retrieves the name of the class (state 432). The system next looks up the class in the list of classes 222 illustrated in FIG. 2 (state 434). Finally, the system returns a class pointer (state 436). An example code listing for the class resolution process appears in Table 2.

Detailed Description Text (26):
After the direct pointer to either the method, interface method, field or class is returned in states 416, 430 and 436, the system replaces the originally accessed constant pool entry with the direct pointer (state 438). The system also modifies the tag (status indicator) associated with the entry to indicate that the entry contains a direct pointer. The above process is repeated for additional constant

pool resolutions.

Detailed Description Text (27):
FIG. 5 is a flow chart illustrating the name and type resolution subroutine in accordance with an embodiment of the present invention. This name and type subroutine is called at states 412 and 424 in the flow chart illustrated in FIG. 4. The system first retrieves a string containing the name (state 502). Next, the system retrieves a type/signature string associated with the name (state 504). Finally, the system returns the name and type (state 508).

Detailed Description Text (29):
FIGS. 6A, 6B and 6C illustrate an example of the constant pool resolution process. FIG. 6A presents a piece of code that defines an object-oriented programming class called "Graphics." The class "Graphics" contains a method called "drawPixel," which takes in two integer parameters "x" and "y," and returns a void value.

Detailed Description Text (30):
The object-oriented code which ultimately calls the method "drawpixel" is translated into a compact representation (known as a bytecode) for execution on platform-independent virtual machines. For example, a program instruction that calls the method "drawPixel" may be compiled into a bytecode in a stream of bytecodes using the JAVA programming environment as is illustrated in FIG. 6B. FIG. 6B illustrates a single instruction. The first byte contains the instruction "Oxb6," which specifies an "invokevirtual" operation, which invokes a method that is specified by a following constant pool index. In the illustrated example, the following constant pool index is a "9," which indicates that the method is specified by the ninth entry of the associated constant pool.

Detailed Description Text (31):
FIG. 6C illustrates the process of resolving the bytecode illustrated in FIG. 6B. More specifically, FIG. 6C contains a constant pool 206, including entries 1 through N, wherein each entry includes a tag, indicating what type of information is stored in the entry, along with the information itself. In the JAVA programming language there are 11 different kinds of constant pool entry types, identified by a one-byte tag field at the beginning of each entry. For instance, there is a specific entry type for storing class, method, interface method, field and string references. The data fields stored after the tag field can vary depending on the type of the constant pool entry. In some entries the data fields contain indices to other constant pool entries. The fact that each constant pool entry may refer to multiple other constant pool entries, which in turn may recursively refer to further entries makes constant pool resolution a time-consuming operation.

Detailed Description Text (32):
The resolution process illustrated in FIG. 6C operates as follows. The bytecode in FIG. 6B specifies that a method reference starting at the ninth entry of the constant pool is to be resolved. The system first examines the tag field of the ninth entry to determine if the entry has already been resolved. If so, the system simply uses the direct pointer stored in the entry to perform the method reference.

Detailed Description Text (33):
If not, the entry must be resolved. In order to resolve the entry, the contents of the ninth entry is first retrieved. In the illustrated example, the ninth entry includes the numbers "4" and "7," which two are additional indexes into constant pool 206. The system then retrieves the fourth entry of constant pool 206, which includes a class identifier along with number "1," which is also an index into constant pool 206. The system next retrieves the first entry of constant pool 206, which contains a pointer to a string containing the name of the class "Graphics."

Detailed Description Text (34):

The system also retrieves the seventh entry of constant pool 206. This entry includes the numbers "2" and "3," which are also indexes in to constant pool 206. The system next retrieves the contents of the second entry, which contains a pointer to a string containing the name of the method, "drawPixel." The system further retrieves the contents of the third entry, which contains a pointer to a string specifying type information for the method. This string "(II)V" indicates that the method takes two integer input parameters and returns a void result. At this point it is possible to perform the method lookup in a related method table to return a direct pointer to the method bytecode.

Detailed Description Text (35):
Finally, the direct pointer is stored in the constant pool entry so that it can be used in future constant pool lookups. In this example, the direct pointer is stored in entry 9. The tag field associated with entry 9 is additionally modified to indicate that the constant pool entry contains a direct pointer.

Detailed Description Paragraph Table (1):
TABLE 1 /*========== * Copyright (c) 1998 Sun Microsystems, Inc. All Rights Reserved * FUNCTION: resolveMethodReference() * TYPE: public instance-level operation * OVERVIEW: Given an index to a CONSTANT_Methodref or * CONSTANT_InterfaceMethodref, get the Method that the index refers to. * INTERFACE: * parameters: constant pool pointer, constant pool index * returns: method pointer *==========*/ METHOD resolveMethodReference(POOLE constantPool, unsigned short cpIndex) { POOLE thisEntry = &constantPool[cpIndex]; METHOD thisMethod = NIL; #if CACHECONSTANTPOOLENTRIES // Check if this entry has already been resolved (cached) // If so, simply return the earlier resolved class if (thisEntry->tag & CP_CACHEBIT) return ((cpCache*)thisEntry->method; #endif // Resolve the class part of the reference short classIndex = ((cpMethodRef*)thisEntry)->classIndex; CLASS thisClass = resolveClassReference(constantPool, classIndex); // Resolve the name and type part short nameTypeIndex = ((cpMethodRef*)thisEntry->nameTypeIndex; char* methodName; // used as return value below char* signature; // ditto getNameAndType (constantPool, nameTypeIndex, methodName, signature); // Perform method lookup on the basis of class, name and type if (thisClass && methodName && signature) { thisMethod = lookupMethod(thisClass, methodName, signature); } #if CACHECONSTANTPOOLENTRIES // Cache the value so that we don't ever have to resolve this entry again if (thisMethod) { thisEntry->tag = CP_CACHEBIT; ((cpCache*) thisEntry)->method = thisMethod; } #endif return thisMethod; }

Detailed Description Paragraph Table (2):
TABLE 2 /*========== * Copyright (c) 1998 Sun Microsystems, Inc. All Rights Reserved. * FUNCTION: resolveclassReference() * TYPE: public instance-level operation * OVERVIEW: Given an index to a CONSTANT_Class, get the class that the index refers to. * INTERFACE: * parameters: constant pool pointer, constant pool index * returns: class pointer *==========*/ CLASS resolveClassReference(POOLE constantPool, unsigned short cpIndex) { POOLE thisEntry = &constantPool[cpIndex]; CLASS thisClass; #if CACHECONSTANTPOOLENTRIES // Check if this entry has already been resolved (cached) // If so, simply return the earlier resolved class if(thisEntry->tag & CP_CACHEBIT) return ((cpCache*)thisEntry)->clazz; #endif // Get class name char* className = getCPUtf8(constantPool, ((cpClass*)thisEntry)->nameindex); // If class is of array type, return java.lang.Object // Otherwise get the referenced class, loading it if necessary if (*className == '[`) thisClass = JavaLangObject; else thisClass = getClass (className); #if CACHECONSTANTPOOLENTRIES // Cache the value so that we don't ever have // to resolve this entry again if(thisClass) { thisEntry->tag .vertline.= CP_CACHEBIT; ((cpCache*)thisEntry)->clazz = thisClass; } #endif return thisClass; }

Other Reference Publication (2):
Publication entitled "The Java Virtual Machine Specification", Chapters 4-5, http://java.sun.com/docs/books/vmspec/html/classfile.doc.html.

Other Reference Publication (3):
Sun Microsystems, Inc.: "Java Virtual Machine Specification 1.0" http://www.sunsite.ee/java/vmspec.1995, XP002128310 Mountain View, CA 94043-1100, USA paragraph OA.1!.

Other Reference Publication (4):
Venners, Bill: "Inside the Java 2 Virtual Machine (Enterprise Computing)" http://www.artima.com/insidejvm/blurb.html, Dec. 17, 1999, XP002128311 p. 2, line 14-line 33 p. 12, line 35--p. 14, line 11 & Venners, Bill: Inside the Java Virtual Machine (Java Master Series) Dec. 1997, McGraw-Hill, USA.

CLAIMS:

1. A method for increasing performance of code executing on a virtual machine, comprising: receiving a class file for a class including a constant pool; executing an instruction on the virtual machine that generates a request to resolve an entry in the constant pool, wherein resolving the entry requires at least one lookup in the constant pool; examining a status indicator associated with the entry to determine if the entry has previously been resolved; if the entry has been previously resolved, returning a pointer to a structure associated with the entry, which was returned during a previous resolution of the entry; and if the entry has not been previously resolved, resolving the entry through at least one lookup to the constant pool to produce a pointer to the structure associated with the entry, replacing the entry in the constant pool with the pointer, wherein replacing the entry in the constant pool with the pointer does not require any additional memory, and modifying the status indicator associated with the entry to indicate that the entry contains a pointer.

2. A method for increasing performance of code executing on a virtual machine, comprising: receiving a request to resolve an entry in a symbol table associated with a class, wherein resolving the entry requires at least one lookup in the symbol table; determining if the entry has previously been resolved; if the entry has been previously resolved, returning a pointer to a structure associated with the entry, which was returned during a previous resolution of the entry; and if the entry has not been previously resolved, resolving the entry through at least one lookup to the symbol table to produce a pointer to the structure, and replacing the entry with the pointer, wherein replacing the entry with the pointer does not require any additional memory.

3. The method of claim 2, wherein the symbol table includes a constant pool, which is part of a class file of a platform independent program.

4. The method of claim 2, further comprising receiving the class file from a remote node across a communication channel.

5. The method of claim 4, further comprising decoupling a platform on which the virtual machine operates from the communication channel prior to receiving the request to resolve the entry in the symbol table.

6. The method of claim 2, wherein the act of resolving the entry includes performing a method lookup that returns a pointer to a method associated with the class.

7. The method of claim 2, wherein the act of resolving the entry includes performing a field lookup that returns a pointer to a field associated with the class.

8. The method of claim 2, wherein the act of resolving the entry includes performing a class lookup that returns a pointer to the class.

9. The method of claim 2, further comprising executing an instruction on the virtual machine that generates the request to lookup an entry in the symbol table.

10. The method of claim 9, wherein executing the instruction includes executing a bytecode of a platform independent program.

11. The method of claim 2, further comprising, if the entry is replaced with the pointer, modifying a status indicator associated with the entry to indicate that the entry contains the pointer.

12. The method of claim 2, wherein determining if the entry has previously been resolved includes examining a status indicator associated with the entry.

13. A computer readable storage medium storing instructions that when executed by a computer cause the computer to perform a method for increasing performance of code executing on a virtual machine, comprising: receiving a request to resolve an entry in a constant pool within a class file for a class, wherein resolving the entry requires at least one lookup in the constant pool; determining if the entry has previously been resolved; if the entry has been previously resolved, returning a pointer to a structure associated with the entry, which was returned during a previous resolution of the entry; and if the entry has not been previously resolved, resolving the entry through at least one lookup in the constant pool to produce a pointer to the structure, and replacing the entry with the pointer, wherein replacing the entry with the pointer does not require any additional memory.

14. An apparatus for increasing performance of code executing on a virtual machine, comprising: a lookup mechanism that looks up an entry in a constant pool within a class file for a class, wherein resolving the entry may require at least one lookup in the constant pool; a status indicator, associated with the entry, which indicates if the entry has previously been resolved; a bypassing mechanism, that returns a pointer to a structure associated with the entry if the entry has been previously resolved; and a resolution mechanism that resolves the entry, if the entry has not been previously resolved, through at least one lookup to the constant pool to produce a pointer to the structure, and that replaces the entry with the pointer, wherein replacing the entry with the pointer does not require any additional memory.

15. The apparatus of claim 14, wherein the constant pool comprises a portion of a class file of a platform independent program.

16. The apparatus of claim 14, further comprising a communication mechanism that receives the class file from a remote node across a communication channel.

17. The method of claim 16, wherein the communication mechanism is configured so that it can be decoupled from the communication channel while code is executing on the virtual machine.

18. The apparatus of claim 14, wherein the resolution mechanism is configured to perform a method lookup that returns a pointer to a method associated with the class.

19. The apparatus of claim 14 wherein the resolution mechanism is configured to perform a field lookup that returns a pointer to a field associated with the class.

20. The apparatus of claim 14, wherein the resolution mechanism is configured to perform a class lookup that returns a pointer to the class.

21. The apparatus of claim 14, further comprising an execution mechanism that executes an instruction on the virtual machine that generates the request to lookup the entry in the constant pool.

23. The apparatus of claim 14, wherein the resolution mechanism is configured to modify the status indicator to indicate that the entry contains a pointer.

24. An apparatus for increasing performance of code executing on a virtual machine, comprising: a lookup means, for looking up an entry in a constant pool within a class file for a class, wherein resolving the entry may require at least one lookup in the constant pool; a status indicator means, associated with the entry, for indicating if the entry has previously been resolved; a bypassing means for bypassing the lookup to the entry by returning a pointer to a structure associated with the entry if the entry has been previously resolved; and a resolution means, for resolving the entry, if the entry has not been previously resolved, through at least one lookup to the constant pool to produce a pointer to the structure, and replacing the entry with the pointer, wherein replacing the entry with the pointer does not require any additional memory.

NO Constant Pool

First Hit Fwd Refs

End of Result Set

☐ Generate Collection Print

L51: Entry 1 of 1 File: USPT Dec 17, 2002

DOCUMENT-IDENTIFIER: US 6496871 B1
TITLE: Distributed agent software system and method having enhanced process mobility and communication in a computer network

Brief Summary Text (10):
RPC provides a way of breaking a program into discrete parts, each of which runs in its own address space. Unlike DSM, RPC communication is explicit in the program, so programmers have complete control over costs. However, the semantics of RPC are substantially different from that of an ordinary procedure call. In particular, when a procedure P makes an RPC call to a procedure Q, the arguments to Q are marshaled and shipped to the machine where the computation should be performed. Stub generators on procedures linked to the application program are responsible for handling representation conversion and messaging. Arguments passed to a remote procedure are passed by copying. Thus, side effects to shared structures can no longer be used for communication between caller and callee. As a result, imperative programs must be substantially modified to run in a distributed environment using RPC. Consequently, programming a distributed agent system using RPC semantics is significantly more complex and subtle than sequential programming on a serial machine.

Brief Summary Text (13):
Recently, a shift to a new computational paradigm has occurred. Instead of regarding the locus of an executing program as a single address space physically resident on a single processor, or as a collection of independent programs distributed among a set of processors, the advent of concurrent, network-centric, object-based languages, such as Java, has offered a compelling alternative. See J. Gosling et al., The Java Language Specification, Sun Microsystems, Inc. (1995), which is expressly incorporated herein. By allowing concurrent threads of control to execute on top of a portable, distributed virtual machine, a network-aware language like Java presents a view of computation in which a single program can be seamlessly distributed among a collection of heterogeneous processors. Unlike distributed systems that require the same code to be resident on all machines prior to execution, code-mobile languages like Java allow new code to be transmitted and linked to an already executing process. This feature allows dynamic upload functionality in ways not possible in traditional distributed systems.

Brief Summary Text (14):
Java incorporates computational units known as "objects." An object includes a collection of data called instances variables, and a set of operations called methods which operate on the instance variables. Object state (i.e. the instance variables) is accessed and manipulated from outside an object through publicly visible methods. Because this objectoriented paradigm provides a natural form of encapsulation, it is generally well-suited for a distributed environment. Objects provide regulated access to shared resources and services. In contrast to distributed glue languages, distributed extensions of Java permit objects as well as base types to be communicated. Moreover, certain implementations, such as Java/RMI, also permit code to be dynamically linked into an address space on a remote site.

h e b b g e e e f c e g g e ge

Brief Summary Text (27):
Generally speaking, in accordance with the invention, a distributed software system for use with a plurality of computer machines connected as a network is provided. The system may comprise a plurality of bases, each base providing a local address space and computer resources on one of a plurality of computer machines. At least one agent comprising a protection domain is provided, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases. A plurality of objects are contained within the protection domain of the at least one agent, a first object residing on a first base of the plurality of bases and a second object residing on a second base of the plurality of bases. The first object on the first base may access the second object on the second base without knowledge of the physical address of the second object on the second base. Finally, at least one runtime system is connected to the first base and the second base. The runtime system facilitates migration of agents and objects from at least the first base to at least the second base.

Brief Summary Text (35):
A still further object of the present invention is to provide a distributed agent system which allows easy and efficient runtime process migration, in whole or in part, among distinct machines.

Drawing Description Text (14):
FIG. 9 schematic diagram showing the relationship of runtime systems to the other basic components of a distributed agent system according to the present invention;

Drawing Description Text (15):
FIG. 10 schematic diagram showing subcomponents of bases, subagents and runtime systems of a distributed agent system according to the present invention;

Drawing Description Text (16):
FIGS. 11A-11E are schematic diagrams showing the relationship of runtime systems of the present invention to an exemplary sequence of agent migration in a distributed agent system according to the present invention;

Drawing Description Text (17):
FIGS. 12A and 12B are schematic diagrams showing the relationship of runtime systems of the present invention to an example of partial agent migration in a distributed agent system according to the present invention;

Drawing Description Text (18):
FIGS. 13A and 13B are schematic diagrams showing the relationship of runtime systems of the present invention to an example of whole agent migration in a distributed agent system according to the present invention;

Drawing Description Text (19):
FIGS. 14A and 14B are schematic diagrams showing the relationship of runtime systems of the present invention to an example of object migration in a distributed agent system according to the present invention;

Detailed Description Text (10):
Each agent 40 is a mobile software component that serves as both a global object space and a protection domain. An agent 40 manages a set of objects that all reside in the same consistent and unique object space. Each agent 40 encapsulates a collection of objects, including simple objects (such as data objects) as well as a collection of threads or concurrently executing tasks. Each agent 40 runs on one or more bases 30, and several agents 40 or several parts of agents may simultaneously reside on the same base 30. Thus, agents of the present invention may reside on one or more bases and also on one or more distinct machines. Accordingly, an agent's state may itself be distributed over a collection of distinct machines. The details

of components necessary to implement such agents are described below in the section entitled "Runtime Data Structure."

Detailed Description Text (24):
A particularly useful feature of the present invention is program mobility. The distributed agent system of the present invention incorporates several user-level migration methods for agents and objects, and one system-level migration method for threads. Migration is important to the invention since it is an important means by which mobility is realized. Unlike other agent systems, the present invention, as a consequence of its distributed agent metaphor, allows any object, and not just agents, to move freely about a network ensemble during runtime. Thus, mobility is realized at both the agent and the object level. Tasks and data may freely and dynamically migrate among the machines in the network associated with creating their agent. By allowing objects and agents to migrate, the invention provides a degree of adaptability and flexibility heretofore unachieved by the prior art.

Detailed Description Text (25):
Agent migration causes an entire agent of the present invention to be moved in a single atomic step. When an agent consists of multiple threads executing on different bases and an agent migration method is called, all of these threads are preferably gathered at one of the bases before migrating to the target base. Object migration permits internal data and associated threads to migrate. (Further details are provided below in the section entitled "Runtime System.")

Detailed Description Text (47):
Programmers may also explicitly specify a base where an invoker method call should be executed using a method name with an `@ [base]` expression, although the invoker method is executed in a caller base by default. In this call, the caller base, the base on which the instance resides, and the base on which the method is executed might be different, but this does not raise an error since the instance methods and fields are always accessed using the slow access mode.

Detailed Description Text (50):
An agent of the present invention communicates with another agent by invoking interface methods of remote objects, for example in much the same way as Java's RMI. First, an interface that extends a Remote interface is defined with the signatures of instance methods that may be called from remote agents. Second, a class that implements the above interface is defined with the implementation of the methods. Third, an instance object created from the above class is recorded in either the agent's own registry or the global registry agent. Fourth, a remote agent looks up an object in the registry and receives the object reference to the actual object. Finally, the remote agent may call the instance method of the remote object in the RPC model, which is always applied to the remote method calls across agents.

Detailed Description Text (52):
Dynamic Linking

Detailed Description Text (53):
The present invention allows new class definitions (i.e. code) to be dynamically injected into a running program. This feature allows applications to incrementally enhance their functionality without requiring their reexecution. The structure of the class loader that provides this feature is similar to related mechanisms found in other languages that provide dynamic linking of new code structures (e.g. Scheme or Java). However, the introduction of a distributed object space raises issues not relevant in previous work.

Detailed Description Text (54):
Due to the distributed object semantics of the present invention, an agent has more than one class loader that control how and where to load classes. The first class

loader that is created at the beginning of execution is preferably linked to the base where the agent starts to run, so that user-defined classes are loaded from the base by default. However, when an object migrates to a new base where the object's class has not yet been loaded, the class cannot be loaded from this new base but must be transferred from the source base (on which the object was originally created) to the destination base.

Detailed Description Text (57):
FIG. 8A shows migration of an object in core library classes. Core class libraries may be considered as representing system classes not modifiable by the programmer. The core class files may always be loaded from a local disk. As shown in FIG. 8A, Base1 holds a class loader and a class object (Class1) and an instance of this class. When this instance moves to Base2, the class file containing the code for this object's methods must be loaded. To do so, the following steps are performed. After the object migrates (arrow A), a remote reference to the defining class found on Base1 is created on Base2 (arrow B). Similarly, a remote reference to the class loader is also created (arrow C). Since Class1 is a core class, it can be loaded from a disk local to the machine on which Base1 is found (arrow D) and linked to the instance of Class1 (arrow E).

Detailed Description Text (58):
FIG. 8B depicts object migration in a user-defined class. In this case, the class file must be loaded from the disk in which the class file exists, or from the source base of the migration, because the source base must have the class file. FIG. 8B illustrates the latter example. Base1 holds a class loader and a class object (Class2) and an instance of this class. When the Class2 instance migrates to Base2 (arrow A), remote references to the class object (arrow B) and the class loader (arrow C) are established. The remote reference to the class loader (arrow C) allows future dynamic linking of class files created on Base1 to be transparently loaded onto Base2. The remote reference to the class object (arrow B) is required because the class object may hold global state information. The class file containing method definitions is then copied (arrow D) from Base1 to Base2. The instance object is then linked to this class file (arrow E).

Detailed Description Text (60):
Runtime System

Detailed Description Text (61):
The runtime system manages a data structure of a base and provides special functions described in this invention by the inventors. As FIG. 9 shows, each base is attached to a corresponding runtime system that provides certain management functions and communication support. A single communication system may be used to serve all of the runtime systems on a particular machine. An agent may comprise a plurality of subagents, each of which resides on separate bases. In this case, subagents in the separate bases are connected with the communication system supported by the runtime system or systems found on their respective bases.

Detailed Description Text (62):
FIG. 10 depicts subcomponents in a base and its runtime system in detail. A base includes a plurality of data blocks, including class files, object memory, task memory and subagent control storage. The object memory stores all objects in a subagent, including reference objects that refer to remote objects outside the subagent. The object memory is managed by an object manager in a runtime system and pointed to by an object table in the subagent control storage. The task memory stores thread frames, used by the task executer to manage task execution. Class files hold programming code that is accessed by the task executer. The subagent control storage stores management information for a subagent. An agent ID in the subagent control storage identifies the specific agent to which the subagent belongs (that is, the agent of which the subagent is a part). An object table in the subagent control storage points to an object memory in the subagent. A task

stack in the subagent control storage points to a task memory to maintain the subagent's execution states.

Detailed Description Text (64):
The task executer also communicates with a task serializer, to which the executer makes requests to serialize task objects, and a remote access controller, to which the executer makes requests to call remote methods. Details of one implementation of the remote access controller are described below (in the section entitled "Runtime Data Structure"). An object manager implements the object space discussed above by managing objects in the object memory, and in particular by instantiating objects, reclaiming garbage objects, and making requests for serializing objects to an object serializer. A communication system mediates interaction among bases in machines connected to the network.

Detailed Description Text (65):
While agent and object migration issues have been generally discussed above (see section entitled "Agent and Object Migration"), the participation of the runtime system in such migration is highlighted in the following additional discussion.

Detailed Description Text (67):
After the Machine B communication system receives the agent migration data for Subagent A (including the agent control data, serialized tasks and serialized objects), a Base B agent manager allocates a memory block for Subagent A on Base B (denoted Subagent A'), and creates a subagent control storage on Base B for Subagent A'. Machine B task executer and object manager also create task objects and data objects in Base B task memory and object memory, respectively. After Subagent A' is thus instantiated on Base B, a class request is sent from Base B to Base A over the network as shown in FIG. 11C. As shown in FIG. 11D, Base A responds to the class request by sending over the network to Base B class files for the agent which are necessary for resuming the agent on Base B. After all migration steps are finished, the memory block for the Subagent A on Base A is released, and the agent resumes as Subagent A' on the Base B as shown in FIG. 11E. In this example, Machine A and Machine B may be heterogeneous. Machine dependencies in the structure of tasks and objects are resolved by the runtime system, and in particular by the task and object serializers.

Detailed Description Text (70):
FIGS. 14A and 14B show an example of object migration. An Agent comprising a single subagent, Subagent A1 resides on a Base A as shown in FIG. 14A. Subagent A1 includes an object memory containing an object, Object O1. A programmer requests that Object O1 migrate from Base A to Base B. Object O1 is serialized and sent to Base B using the Base A runtime system and communication system. The Base B communication system receives the serialized Object O1, and if there is no subagent associated with the Agent on Base B, then the Base B runtime system creates a new memory block for a new subagent, Subagent A2, as shown in FIG. 14B. In this example, the Agent resides on both Base A and Base B after the migration of Object O1, and a forwarding object "r" is created in Subagent A1' to Object O1 on Base B to maintain network-transparent references to Object O1 even after the object migration.

Detailed Description Text (73):
Runtime Data Structure

Detailed Description Text (89):
Threads express execution states of programs in runtime and may be instances of a thread class. In addition to the standard fields for that class, each thread contains a stack. This stack is a linked list of stack segments, each of which contain a sequence of stack frames. An implementation of a frame contains a pointer to size and type information for local variables and arguments. This information is used to properly handle routine type checking, and is also used by the garbage

collector. It is possible to evaluate this information dynamically if garbage collection occurs infrequently.

CLAIMS:

1. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising: a plurality of bases, each base providing a local address space and computer resources on one of the plurality of computer machines; at least one agent comprising a protection domain and a global object space, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases; a plurality of objects contained within the protection domain of the at least one agent, a first object residing on a first base of the plurality of bases and a second object residing on a second base of the plurality of bases, wherein the global object space includes a mapping of symbolic references of objects within the at least one agent to corresponding physical addresses of said objects, whereby the first object on the first base may access the second object on the second base without knowledge of the physical address of the second object on the second base by obtaining a symbolic reference to the second object from the first object and obtaining the corresponding physical address of the second object using the mapping of the global object space; and at least one runtime system connected to the first base and the second base, the at least one runtime system facilitating migration of agents and objects from at least the first base to at least the second base.

2. The distributed software system of claim 1, wherein each agent further comprises at least one subagent, each subagent residing on one base and comprising: an object memory which stores objects in the subagent; a task memory which stores task frames in the subagent; program code for the agent to which the subagent belongs; a subagent control storage comprising: an agent identifier indicating the agent to which the subagent belongs; an object table having a mapping which maps symbolic references of objects to corresponding physical addresses of said objects in the object memory; a task stack which stores a plurality of task thread pointers in the task memory;

and wherein the at least one runtime system further comprises: an agent manager for each base managing a plurality of subagent control storages of subagents residing on the corresponding base; an object manager for each base managing a plurality of object memories for a plurality of subagents residing on the corresponding base; an object serializer for each base serializing objects for transmitting the objects across the network to at least one base other than the corresponding base; a task executer for each base reading program code, creating task stacks in task memories, and executing the program code; a task serializer for each base serializing task stacks for transmitting the stacks across the network to at least one base other than the corresponding base; a remote access controller for each base receiving remote object access messages from a task executer on at least one base other than the corresponding base and sending remote object access requests to at least one base other than the corresponding base; and a communication system coordinating physical communication between the computer machines.

4. The distributed software system of claim 2, wherein the runtime system further facilitates migration of tasks from the first base to the second base.

10. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising: a plurality of bases, each base providing a local address space and computer resources on one of a plurality of computer machines; at least one agent comprising a protection domain, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases; a plurality of objects contained within the protection domain of the at least one agent, a first object residing on a first base of the plurality of bases and a second object residing on a second base of the plurality of bases,

wherein the first object on the first base may access the second object on the second base without knowledge of the physical address of the second object on the second base, wherein the access by the first object of the second object is a method call specifying at least one of an argument and a return value, wherein a symbolic reference to the at least one argument or return value may be passed to or returned from the called method to identify the at least one argument or return value, and wherein the physical address of the at least one argument or return value need not be passed to or returned from the called method to identify the at least one argunent or return value so as to render the method call network transparent; and at least one runtime system connected to the first base and the second base, the at least one runtime system facilitating migration of agents and objects from at least the first base to at least the second base.

25. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising: a plurality of bases, each base providing a local address space and computer resources on one of the plurality of computer machines; at least one agent residing on a first base and comprising a protection domain, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases; at least one object residing within the protection domain of the at least one agent and including at least one anchored object, the at least one anchored object being instantiated on the first base from a base-dependent class and which is permanently unable to be moved from the first base to any other base; and at least one runtime system connected to the plurality of bases, the at least one runtime system including a communication system which facilitates migration of agents and objects among the plurality of bases.

26. The distributed software system of claim 25, wherein each agent further comprises at least one subagent, each subagent residing on one base and comprising: an object memory which stores objects in the subagent; a task memory which stores task frames in the subagent; program code for the agent to which the subagent belongs; a subagent control storage comprising: an agent identifier indicating the agent to which the subagent belongs; an object table having a mapping which maps symbolic references of objects to corresponding physical addresses of said objects in the object memory; a task stack which stores a plurality of task thread pointers in the task memory;

and wherein the at least one runtime system further comprises: an agent manager for each base managing a plurality of subagent control storages of subagents residing on the corresponding base; an object manager for each base managing a plurality of object memories for a plurality of subagents residing on the corresponding base; an object serializer for each base serializing objects for transmitting the objects across the network to at least one base other than the corresponding base; a task executer for each base reading program code, creating task stacks in task memories, and executing the program code; a task serializer for each base serializing task stacks for transmitting the stacks across the network to at least one base other than the corresponding base; a remote access controller for each base receiving remote object access messages from a task executer on at least one base other than the corresponding base and sending remote object access requests to at least one base other than the corresponding base; and a communication system coordinating physical communication between the computer machines.

28. The distributed software system of claim 26, wherein the runtime system further facilitates migration of tasks among the plurality of bases.

40. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising: a plurality of bases, each base providing a local address space and computer resources on one of the plurality of computer machines; at least one agent comprising a protection domain, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases; at least one object residing within the protection domain of

the at least one agent; at least one runtime system connected to the plurality of bases, the at least one runtime system including a communication system which facilitates migration of agents and objects among the plurality of bases; a first method calling protocol for calling, from a first base, a method to a first object residing on a second base, wherein the method is transmitted from the first base to the second base and wherein the method is executed on the second base where the first object resides; and a second method calling protocol for calling, from the first base, a method to a second object residing on the second base, wherein the method is executed on the first base using method code on the first base corresponding to the second object method and a remote reference to the second object on the second base.

55. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising: a plurality of bases, each base providing a local address space and computer resources on one of the plurality of computer machines; at least one agent residing on a first base and comprising a protection domain, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases; at least one object residing within the protection domain of the at least one agent and including at least one pinned object which is temporarily unable to be moved from the first base to any other base; and at least one runtime system connected to the plurality of bases, the at least one runtime system including a communication system which facilitates migration of agents and objects among the plurality of bases.

56. The distributed software system of claim 55, wherein each agent further comprises at least one subagent, each subagent residing on one base and comprising: an object memory which stores objects in the subagent; a task memory which stores task frames in the subagent; program code for the agent to which the subagent belongs; a subagent control storage comprising: an agent identifier indicating the agent to which the subagent belongs; an object table having a mapping which maps symbolic references of objects to corresponding physical addresses of said objects in the object memory; a task stack which stores a plurality of task thread pointers in the task memory;

and wherein the at least one runtime system further comprises: an agent manager for each base managing a plurality of subagent control storages of subagents residing on the corresponding base; an object manager for each base managing a plurality of object memories for a plurality of subagents residing on the corresponding base; an object serializer for each base serializing objects for transmitting the objects across the network to at least one base other than the corresponding base; a task executer for each base reading program code, creating task stacks in task memories, and executing the program code; a task serializer for each base serializing task stacks for transmitting the stacks across the network to at least one base other than the corresponding base; a remote access controller for each base receiving remote object access messages from a task executer on at least one base other than the corresponding base and sending remote object access requests to at least one base other than the corresponding base; and a communication system coordinating physical conununication between the computer machines.

58. The distributed software system of claim 56, wherein the runtime system further facilitates migration of tasks among the plurality of bases.

use Constant Pool Generally

DOCUMENT-IDENTIFIER: US 6003038 A
TITLE: Object-oriented processor architecture and operating method

Abstract Text (1):
A class structure of an object-oriented program system is optimized for hardware and implemented as a hardware system in an object-oriented processor. For example, a class structure derived from a Java Virtual Machine software system is optimized for hardware and implemented as a hardware Java object-oriented processor. A processor is implemented by defining a data structure and an object-oriented instruction set for executing in the object-oriented operating environment, and generating a hardware implementation of the processor enforcing the logical relationships of the instruction set as defined by the data structure. The data structure includes a class structure. The class structure and the instruction set describe the processor operations. A processor includes an execution engine based on a class structure to execute instructions of an object-oriented instruction set. The instruction set uses pointers for indexing through data structures to define an object method for execution. At the top of the data structure is a class structure for selecting the static class for creating objects. Once the class is found by indexing through the class structure, a constant pool is accessed to supply information to further describe the data structures that are used for creating an object and executing an object method. A processor includes an execution unit and a memory. The memory stores a data structure including a class structure. Classifiers are selected by a program and loaded. The class structure directs the instantiation of the loaded classifiers into objects. The execution unit executes object methods directed by the class structure.

Brief Summary Text (3):
The present invention relates to processors and computer systems. More specifically, the present invention relates to an object-oriented processor architecture and operating method.

Brief Summary Text (6):
Software technologies have evolved to advantageously employ a concept of object-oriented computing in which program operations are organized as cooperative collections of objects, each representing an instance of some class. The classes are members of a hierarchy of classes that are united by inheritance relationships. In the object-oriented structure, classes are generally viewed as static while objects have a dynamic nature. Object-oriented approaches have led to improved productivity in program production, improved program reliability and reduction of programming errors, and improved performance in complex programs. Object-oriented approaches are universally applied as software constructs such as purely C++ type data structures that are suitable for software execution and implementation at the very high application level.

Brief Summary Text (9):
What is needed is a processor and associated operating method that achieves object-oriented functionality without incurring software overhead and degraded performance.

h e b b g e e e f c e g g e ge

Brief Summary Text (11):
Object-oriented computing operates on the basis of static classes that create dynamic objects when the objects are needed and terminating the objects when the objects are idle. The object-oriented processor attains object-oriented computing in a hardware processor by defining data structures and implementing an instruction set based on the defined data structures. In the object-oriented processor, a central processing unit (CPU) executes by creating an object, computing a method inside the created object and during the computation creating additional objects, and jumping to the additional objects. The CPU accordingly moves from object to object, creating new objects when new objects are needed and selectively terminating objects when the objects are no longer executing.

Brief Summary Text (12):
In accordance with an aspect of the present invention, a class structure of an object-oriented program system is optimized for hardware and implemented as a hardware system in an object-oriented processor. In a particular embodiment, a class structure derived from a Java.TM. Virtual Machine software system is optimized for hardware and implemented as a hardware Java.TM. object-oriented processor.

Brief Summary Text (13):
In accordance with an aspect of the present invention, the structure and operations of a processor are precisely defined and implemented by defining a data structure of an object-oriented operating environment, defining an object-oriented instruction set for executing in the object-oriented operating environment, and generating a hardware implementation of the processor enforcing the logical relationships of the instruction set as defined by the data structure. The data structure includes a class structure. The class structure and the instruction set unambiguously and precisely describe the operations of the processor.

Brief Summary Text (14):
In accordance with an aspect of the present invention, a processor includes a plurality of registers for directing the execution of object methods according to a predetermined data structure design. A memory includes data structures, including a class structure, that are defined for objects that are to be executed by the processor. The memory implements an object-oriented instruction set in accordance with the data structures. An execution unit executes object methods according to the object-oriented instruction set implemented in the memory.

Brief Summary Text (15):
In accordance with an aspect of the present invention, a processor includes an execution engine that is organized based on a class structure to execute a plurality of instructions of an object-oriented instruction set. The instruction set is executed using pointers for indexing through a plurality of data structures to define an object method for execution. At the top of the data structure is a class structure for selecting the static class for creating objects. Once the class is found by indexing through the class structure, an additional level of the data structure called a constant pool is accessed to supply information to further describe the data structures that are used for creating an object and executing an object method.

Brief Summary Text (16):
In accordance with an aspect of the present invention, a processor includes an execution unit and a memory. The memory stores a data structure including a class structure. A plurality of classifiers is selected by a program and loaded. The class structure directs the instantiation of the loaded classifiers into objects. The execution unit executes object methods directed by the class structure stored in the memory.

Brief Summary Text (17):
In accordance with an embodiment of the present invention, an object-oriented processor includes a memory storing a data structure including a class structure defined on an object-oriented basis and an execution unit connected to the memory including a logic for deriving a pointer into the class structure, the pointer designating a class for instantiating an object and executing the object as a method.

Brief Summary Text (18):
Many advantages are attained by the described object-oriented processor and operating method. It is advantageous that the advantages of object-oriented functionality are gained without incurring degradation in performance resulting from imposition of software overhead. It is advantageous that the advantages of object-oriented functionality are gained prior to software initialization. It is advantageous that object-oriented functionality is attained immediately upon hardware power-up so that faults in the software initialization process do not impair system functionality. It is advantageous that the implementation of object-oriented functionality in hardware reduces the complexity of a system.

Drawing Description Text (2):
The features of the described embodiments believed to be novel are specifically set forth in the appended claims. However, embodiments of the invention relating to both structure and method of operation, may best be understood by referring to the following description and accompanying drawings.

Drawing Description Text (3):
FIG. 1 is a block diagram of an embodiment of a virtual machine hardware processor.

Drawing Description Text (5):
FIG. 3 is a schematic data structure diagram depicting a class structure which is derived from the data structures of a Java.TM. Virtual Machine implementation.

Drawing Description Text (6):
FIG. 4 is a schematic data structure diagram showing an embodiment of a constant pool, an array with four-byte entries used for storing constant data.

Drawing Description Text (7):
FIG. 5 is a schematic data structure diagram showing a field array layout including a collection of the fields defined in a current class.

Drawing Description Text (8):
FIG. 6, is a schematic data structure diagram illustrating a method array for storing methods of a class.

Detailed Description Text (2):
FIG. 1 illustrates one embodiment of a virtual machine instruction hardware processor 100, hereinafter hardware processor 100, in accordance with the present invention, and that directly executes virtual machine instructions that are processor architecture independent. The performance of hardware processor 100 in executing JAVA.TM. virtual machine instructions is much better than high-end CPUs, such as the Intel PENTIUM microprocessor or the Sun Microsystems ULTRASPARC processor, (ULTRASPARC is a trademark of Sun Microsystems of Mountain View, Calif., and PENTIUM is a trademark of Intel Corp. of Sunnyvale, Calif.) interpreting the same virtual machine instructions with a software JAVA.TM. interpreter or with a JAVA.TM. just-in-time compiler; is low cost; and exhibits low power consumption. As a result, hardware processor 100 is well suited for portable applications. Hardware processor 100 provides similar advantages for other virtual machine stack-based architectures as well as for virtual machines utilizing features such as garbage collection and thread synchronization.

Detailed Description Text (3):
In view of these characteristics, a system based on hardware processor 100 presents an attractive price for performance characteristics, if not the best overall performance, as compared with alternative virtual machine execution environments including software interpreters and just-in-time compilers. The illustrative embodiments apply to hardware implementing the JAVA.TM. virtual machine in microcode, directly in silicon, or in some combination thereof.

Detailed Description Text (4):
As used herein, the illustrative hardware machine implements instructions of the Java.TM. virtual machine where a virtual machine is an abstract computing machine that, like a real computing machine, has an instruction set and uses various memory areas. A virtual machine specification defines a set of processor architecture independent virtual machine instructions that are executed by a virtual machine implementation such as hardware processor 100. Each virtual machine instruction defines a specific operation that is to be performed, The virtual computing machine need not understand the computer language that is used to generate virtual machine instructions or the underlying implementation of the virtual machine. Only a particular file format for virtual machine instructions need to be understood.

Detailed Description Text (5):
Thus in an exemplary embodiment, the implemented machine instructions are JAVA.TM. virtual machine instructions. Each JAVA.TM. virtual machine instruction includes one or more bytes that encode instruction identifying information, operands, and any other required information.

Detailed Description Text (6):
In this embodiment, hardware processor 100 processes the JAVA.TM. virtual machine instructions, which include bytecodes. Hardware processor 100 executes directly most of the bytecodes. However, execution of some of the bytecodes is implemented via microcode.

Detailed Description Text (9):
Referring to FIG. 2, a schematic block diagram illustrates an object-oriented processor 202 connected to a memory 204. The object-oriented processor 202 functions on an object-oriented basis of a class structure that is defined in terms of a memory layout. With the defined class structure, the object-oriented processor 202 derives a class pointer 208 and uses the class pointer 208 to create objects as instances of classes and execute the created objects. A class pointer 208 may be derived for each creation of an object or advantageously stored for efficient multiple instantiation of objects. A compiler is typically used to determine whether a particular class pointer is to be saved for multiple instantiation of objects. Because an object is the instantiation of a class, object information held in a constant pool 206 in the memory 204 is very simple. The object structure is stored as an array in the constant pool 206 as a block of condensed information of class structure. In the illustrative embodiment each class has an associated constant pool 206.

Detailed Description Text (10):
The object-oriented processor 202 implements instructions in the instruction set using direct access and indirect access of data in the data structures with variable levels of accessing the data structures. For example, the data structure in the memory 204 includes storage elements holding addresses for direct access in which hardware reads the address and directly accesses the address to retrieve data. In an indirect access, the data structure in the memory 204 includes storage elements holding a pointer to another storage element in the structure, the additional storage element holds an address that is retrieved using the pointer. The address is accessed and data is retrieved in the indirect access. For some objects, the data structure includes multiple levels of indirect addressing to

access information. The class structure defines the number of levels of access. The class structure defines the objects for generation by the object-oriented processor 202. Some instructions direct that the object-oriented processor 202 jump from one object to another object with the class structure defining the destination of the jump operation.

Detailed Description Text (11):
In an illustrative embodiment, the class structure for the object-oriented processor 202 is defined by the existing Java.TM. Virtual Machine for Javasoft.TM.. Existing processors available from Sun Microsystems, Inc. use the Java.TM. Virtual Machine class structure but the existing processor hardware is not optimized for the class structure and the instruction set. The existing processor implements the class structure generally in software so that advantages of the structure are attained only for high level operations or application level operations. In contrast, the illustrative object-oriented processor 202 gains advantages at the fundamental hardware level by integrating the class structure and the instruction set directly into the hardware design so that the class structure dictates the fundamental hardware operations of the object-oriented processor 202.

Detailed Description Text (12):
The illustrative object-oriented processor 202 has a hardware structure that uses multiple-level memory accesses to index each part of a multiple-part instruction and executing a method directed by the memory accesses using a technique such as trap emulation, microcode sequencing, or hard-coded hardware execution. The object-oriented processor 202 utilizes a relatively complex multiple-level data structure and multiple function instruction execution. For example, instruction execution includes accessing of multiple indices, adding of a constant pool location value to the indices to set a memory address, loading from the address, acquiring the content of the memory location. In some instruction embodiments the content of the memory location gives two indices including a class index, and a name and type index. The class index serves as basis for setting a class pointer. The class pointer designates a class for instantiating objects. The object-oriented processor 202 is substantially different from a conventional RISC processor which executes simple, individual operations in which most accesses are either a load operation or a store operation.

Detailed Description Text (13):
Referring to FIG. 3, a schematic block diagram depicts a class structure 300 which is derived from the data structures of a Java.TM. Virtual Machine implementation. The data structures of the object-oriented processor 202 are optimized for execution in hardware. The object-oriented processor 202 operates according to an hierarchical scheme with a class table 302 defining a first level in the hierarchy of the class structure 300. During execution, the object-oriented processor 202 creates new objects by instantiating selected classes, executing methods defined in the classes, creating objects, deleting objects, and the like.

Detailed Description Text (14):
The class structure 300 of the object-oriented processor 202 hierarchically begins with the class table 302. The class table 302 is a list of all possible classes used in the object-oriented processor 202. In the illustrative embodiment, the class contains three blocks, specifically a constant pool 304, fields 306, and methods 308. The constant pool 304 defines constant values used in the class. The fields 306 define variables, either objects or primitives for the class. The methods 308 define a byte code 310 to be executed and exception tables 312 for error handling.

Detailed Description Text (15):
The class table 302 is an array with an unlimited number of entries. In the illustrative embodiment, an entry in the class table 302 is a 32-bit memory address to a class. The last entry is zero to indicate the end of the class table 302. A

class table 302 is useful for accessing all classes in the object-oriented processor 202. Trap handlers use the class table 302 for resolving traps. A power-on sequence uses the class table 302 to find a first class and instantiate the first class to activate the object-oriented processor 202. Error handlers use the class table 302 to locate an exception class and create an exception object from the exception class to process errors and other exception conditions.

Detailed Description Text (16):
The illustrative class structure 300 has eight 32-bit words. Each entry of the class table 302 is a four-byte wide, 32-bit word. A constant pool entry is the address to the constant pool 304 of the current class and a constantPoolCount is the length of the field array. For a particular class, the constantPoolCount is fixed. A fieldAddress entry is the address to the field array 306 of the current class and fieldCount is the length of the field array 306. A methodAddress entry is the address to the method array 308 of the current class and a methodCount entry is the length of the method array 308.

Detailed Description Text (17):
An objHintBlock entry is an address to a block containing useful information such as instance variables, instance methods, and the size of the object in bytes. The objHintBlock entry is useful for creating objects from the class efficiently. Information in the objHintBlock is entered during class resolution and used by various instructions, including a new instruction. A name entry holds the name of the current class and a superName entry holds the name of the superclass of the current class. To reduce memory space, strings defined in the constant pool are reused. Implements and implementsCount entries are applied for usage by interfaces. A value in the imnplements entry is an index to the constant pool table. The type of entry in the constant pool is of class type, which is an interface of Java.TM.. An access entry is a short integer used to hold flags including a public access flag, a final access flag, a super access flag, an interface access flag, and an abstract access flag.

Detailed Description Text (18):
Referring to FIG. 4, the constant pool 304 is an array with four-byte entries used for storing constant data in the form of an integer, a generic pointer, a floating point number or a character pointer. Various stored data include a class, a field, a string, and a primitive. The first entry (cp[0]) in the constant pool 304 is the address to a type table. The length of the type table is the same as the length of the constant pool. Entries in the type table are one byte wide and represent the type of data stored in the corresponding field of the constant pool. The most significant bit of a type table entry is for resolution, indicating by a "1" value that the corresponding constant pool entry is resolved and the content is the data or address to the data. A "0" value of the resolution bit indicates that the constant pool entry is not resolved and the content is still and index to another constant pool entry. Possible types stored in the type table include a non-used entry, a CONSTANT type holding a string preceded by a 32-bit key field, an unused type, an integer type, a floating point type, a long type, a double type, a class type, a string type, a Fieldref type, a Methodref type, an InterfaceMethodref type, and a NameAndType type. A double constant is stored in two entries. MethodRef, FieldRef, NameAndTypeRef, and InterfaceMethodRef are stored as two short integers in one entry. The length in a CONSTANT.sub.-- Utf8 is the length N of the c-string in bytes.

Detailed Description Text (19):
An example describes the content of a constant pool, as follows:

Detailed Description Text (20):
1, Class(0), index 14

Detailed Description Text (21):

2, Class(1), index 1137208 115a38.rarw.resolved class (current)

Detailed Description Text (22):
3, Fieldref, class index 2, nameType index 5

Detailed Description Text (23):
4, Methodref, class index 1, nameType index 6

Detailed Description Text (33):
14, Utf8*, location 11634ah, value java/lang/Object

Detailed Description Text (41):
In the illustrative example, only the second entry is resolved and the original index is replaced by the decaf class. The remaining reference entries are not resolved. During execution, resolution occurs on-the-fly. When a reference is encountered, the original index is replaced with the appropriate data structure and the index is replaced with the real value of the address to the value.

Detailed Description Text (42):
An example of a Type Table in a constant pool 304 is shown as follows using one byte per entry with the most significant bit reserved for resolution.

Detailed Description Text (50):
7 CONSTANT.sub.-- Class

Detailed Description Text (56):
Referring to FIG. 5, the field array 306 is a collection of the fields defined in the current class. The field array 306 is used to store a static variable or an instance variable. A variable can be either an array or an object. The field structure of the field array 306 has four entries including a four-byte class index or class pointer classPtr 504, a string signature 506, an eight-byte content element holding the object 502, and a two-byte access 510 element. The entries in the field array 306 have a fixed length. For a particular entry in the field array 306, the first eight bytes store the value of the entry which is an object 502. The object 502 has one of several forms including an offset for an instance variable, a value for primitive data types, and object pointer objPtr for an array of primitives, an instance of a class or the like. If an object 502 is not a primitive, objPtr may be the address of the object and objSize the length of the object. If the field contains an instance variable, the object field is the field offset of the object instantiated from the class.

Detailed Description Text (57):
The second entry in the field array 306 is a class pointer classPtr 504 entry which specifies the address of the class containing the field. The third entry in the field array 306 is a signature 506 that describes the type of field entry. The signature 506 is a combination of characters including:

Detailed Description Text (65):
SIGNATURE.sub.-- CLASS `L`

Detailed Description Text (70):
For example, a label `[I` designates that a field is an integer array. A label of `Labc` designates that a field is an object of class abc.

Detailed Description Text (71):
The fourth entry in the field array 306 is a name 508 that designates the name of the field. Name 508 is a pointer to a string in the constant pool 304. All strings in the constant pool 304 contain a 32-bit key field at the first location.

Detailed Description Text (80):

h e b b g e e e f c e g g e ge

Referring to FIG. 6, the method array 308 stores methods of a class. A method structure, as defined by the method array 308, includes the byte code 310, the exception tables 312, and several variables describing the byte code 310 and the exception tables 312. The entries in the method array 308 represent one method. The information held in an entry of the method array 308 includes a byte code 310 of the method, an exception table 312, a method field 602, and several variables to facilitate method invocation. The first entry in the method array 308 is the byte code 310, the address to the location of the actual byte-code. The byte-code is simply a memory block in the memory 204. The second entry in the method array 308 is a codeLength 604 designating the length of the byte code in the current method. The program counter starts from 0 for the execution of a method.

Detailed Description Text (81):
The third entry in the method array 308 is the exception tables 312, the address to the exception table. All entries in the exception table contain a longword startPC 606 designating the related starting address of the try block, a longword endPC 608 designating the end address of the try block, a longword hdlPC 610 supplying the related address of the catch clause, and a shortword catchObj 612 supplying the address of the exception object.

Detailed Description Text (82):
The fourth entry of the method array 308 is expTblLength 614, the length of the exception table. The method array 308 also includes a methodField 616 which is a field block having the same layout as an entry of the field array 306. The method signature, method name, and method access are stored in the methodField 616.

Detailed Description Text (83):
Several variables are stored in the method array 308. ArgsSize 618 is the length of the method arguments. MaxStack 620 defines the maximum length of the possible used stack. Nlocals 622 stores the number of local variables.

Detailed Description Text (84):
Referring to FIG. 7, a schematic diagram illustrates an object format with a handle reference to illustrate the operations of the object-oriented processor 202 in response to a trap. An ObjectRef 702 is a pointer to a storage containing either the contents of an object for a "NO.sub.-- HANDLE case" or another pointer to the actual storage for a "HANDLE case". In either case, the immediate storage object 704 that is pointed to by the ObjectRef 702 is associated with a separate pointer to a MethodVector 706, which is a per-class entity. The MethodVector 706 contains pointers to a message block table that is accessible for the target class. The MethodVector 706 also has pointers for the constant pool 304 and the Class Structure. The object storage 704 has a size field 708 which represents the size of the instance fields in 32-bit words. The size field 708 may be zero for an object with no instance fields defined.

Detailed Description Text (86):
A class declaration specifies a new reference type and implements the class as an extension or subclass of an existing class, or as one or more interfaces. The body of a class declares members including fields and methods, static initializers, and constructors.

Detailed Description Text (87):
Fields are the variables of a class type. Class (static) variables exist once per class. Instance variables exist once per instance of the class. Fields include initializers and are modified using various modifier keywords.

Detailed Description Text (88):
A method declares executable code that can be invoked with a fixed number of values passed as arguments. Every method declaration belongs to some class.

Detailed Description Text (89):
A constructor is used in the creation of an object that is an instance of a class.

Detailed Description Text (90):
A Java.TM. Virtual Machine (JVM) signals an error to an executing Java.TM. program as an exception when the program violates the semantic constraints of the Java.TM. language. The exception is thrown upon the occurrence of a violation and causes nonsocial transfer of control from the location of the exception to a predetermined location. An exception is "thrown" from the location of occurrence and "caught" at the predetermined location. Java.TM. programs can throw exceptions explicitly using a "throw" statement An exception is represented as an instance of the class "throwable" or a "throwable" subclass. During the process of throwing an exception, a Java.TM. Virtual Machine abruptly completes any expressions, statements, method and constructor invocations, static initializers, and field initialization expressions that have begun but have not complete execution in the current thread.

Detailed Description Text (91):
Runtime data areas include a program counter (pc) register, a Java.TM. stack, a heap, a method area, a constant pool, and native method stacks. A pc register is associated with each JVM thread that is supported by the Java.TM. Virtual Machine during multiple-thread execution. The pc register contains the address of the Java.TM. Virtual Machine instruction that is currently executed when the executing method is not native. When the method is native, the pc register is undefined. The pc register is one word in width, a width sufficient to hold a returnAddress or a native pointer on the appropriate platform.

Detailed Description Text (92):
The Java.TM. stack is a private stack that is associated to each Java.TM. Virtual Machine thread and created at the creation of the thread. The Java.TM. stack stores Java.TM. Virtual Machine frames. The Java.TM. stack functions as a conventional stack, operating to store local variables and partial results and handling method invocation and return.

Detailed Description Text (93):
The Java.TM. Virtual Machine has a "heap" that is shared among all threads. The heap is a runtime data area from which memory for all class instances and arrays is allocated. The Java.TM. heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system, called a garbage collector. Objects are never explicitly deallocated.

Detailed Description Text (94):
The Java.TM. Virtual Machine has a method area that is shared among all threads. The method area is analogous to the storage area for compiled code of a conventional language. The method area stores pre-class structures including the constant pool, field and method data, and the code for methods and constructors. The method area is created on virtual machine start-up.

Detailed Description Text (95):
The constant pool is a per-class or per-interface runtime representation of the constant.sub.-- pool table in a Java.TM. class file. The constant pool contains several types of constants including numeric literals that are known at compile time and method and field structures that are resolved at runtime. The constant pool operates in the manner of a symbol table for conventional programming languages, but contains a wider range of data than a typical symbol table. Each constant pool is allocated from the Java.TM. Virtual Machine method area. The constant pool for a class or interface is created when a Java.TM. class file for the class or interface is successfully loaded by the Java.TM. Virtual Machine. An OutOfMemoryError exception occurs when a class file is loaded if the creation of the constant pool is larger than the amount of memory that can be made available in the method area of the Java.TM. Virtual Machine.

Detailed Description Text (96):
An implementation of the Java.TM. Virtual Machine may use conventional stacks to support native methods written in languages other than Java.TM.. A native method stack may be used to implement an emulator of the Java.TM. Virtual Machine instruction set in another language.

Detailed Description Text (97):
A Java.TM. Virtual Machine frame is used to store data and partial results, perform dynamic linking, return values for methods, and dispatch exceptions. A new frame is invoked each time a Java.TM. method is invoked. A frame is destroyed when the method completes, whether a normal or abnormal completion. Frames are allocated from the Java.TM. stack of the thread creating the frame. A frame has a set of local variables and an operand stack. The memory space for the frame variables and stack are allocated simultaneously since the size of the local variable area and the operand stack are known at compile time and the size of the frame data structure depends only on the implementation of the Java.TM. Virtual Machine. Only one frame, the "current frame" for the executing "current" method, is active at any point in a given thread of control. Operations on local variables and the operand stack are always with reference to the current frame.

Detailed Description Text (98):
A frame is no longer current when the method invokes another method or the method completes. When a method is invoked, a new frame is created and becomes current when control transfers to the new method. On return from the method, the current frame passes the result from the method invocation to the current method. Java.TM. Virtual Machine frames operate as though allocated on a stack with one stack per Java.TM. thread. A frame created by a thread is only directly referenced by that thread.

Detailed Description Text (99):
On each invocation of a Java method, the Java.TM. Virtual Machine allocates a Java.TM. frame which contains an array of local variables and contains an operand stack. Most Java.TM. Virtual Machine instructions take values from the operand stack of the current frame, operate on the values, and return the results of the operations on the same operand stack. The operand stack is also used to pass arguments to methods and receive results from the methods.

Detailed Description Text (100):
A Java.TM. Virtual Machine frame contains a reference to the constant pool for the type of the current method to support dynamic linking of the method code. The class file code for a method refers to methods to be invoked and variables to be accessed via symbolic references. Dynamic linking translates the symbolic method references into fixed method references, loads classes to resolve undefined symbols, and translates variable accesses into offsets in storage structures associated with runtime location of the variables.

Detailed Description Text (101):
A method invocation completes normally if the invocation does not cause an exception to be thrown, either directly from the Java.TM. Virtual Machine or resulting from execution of an explicit throw statement. Upon normal completion of an invoked method by execution of a return instruction, the invoked method may return a value to the invoking method. The Java.TM. Virtual Machine frame restores the state of the invoking method including restoration of local variables, operand stack, and the program counter incremented past the method invoking instruction.

Detailed Description Text (102):
A method invocation completes abnormally if execution of a Java.TM. Virtual Machine instruction within the method causes the Java.TM. Virtual Machine to throw an exception and the exception is not handled within the method. Evaluation of an

explicit throw statement also causes an exception to be thrown and, if the exception is not caught by the current method, results in abnormal method completion. A method that completes abnormally does not return a value to the invoking method.

Detailed Description Text (103):
At the level of the Java.TM. Virtual Machine, the constructors function as instance initialization methods having a special name <init>, which is supplied by a Java.TM. compiler. The name <init> is not a valid identifier and therefore cannot be used directly by a Java.TM. programmer. Instance initialization methods are invoked within the Java.TM. Virtual Machine using an invokespecial instruction and are invoked only on uninitialized class instances.

Detailed Description Text (104):
At the level of the Java.TM. Virtual Machine, a class or interface is initialized by invoking the appropriate class initialization method or interface initialization method with no arguments having a special name <clinit>, which is supplied by a Java.TM. compiler. The name <clinit> is not a valid identifier and therefore cannot be used directly by a Java.TM. programmer. Class and interface initialization methods are invoked indirectly and implicitly by the Java.TM. Virtual Machine as part of the class initialization process. Class and interface initialization methods are neither invoked directly from Java code nor directly from a Java.TM. Virtual Machine instruction.

Detailed Description Text (105):
In general, throwing of an exception causes an immediate dynamic transfer of control that may exit multiple Java.TM. statements and multiple constructor invocations, static and field initializer evaluations, and method invocations until a catch clause is found that catches the thrown value. If no catch clause is found in the current method, then the current method invocation completes abnormally. The operand stack and local variables are discarded and the frame is popped, reinstating the frame of the invoking method. The exception is then rethrown in the context of the invoker's frame, continuing a chain of method invocations. If no suitable catch clause is found prior to reaching the top of the method invocation chain, execution of the thread that threw the exception is terminated.

Detailed Description Text (106):
At the level of the Java.TM. Virtual Machine, each catch clause describes the Java.TM. Virtual Machine instruction range for which the catch clause is active, describes the types of exceptions to handle, and supplies the address of the exception handling code for the exceptions. An exception matches a catch clause if the instruction that caused the exception is in an appropriate instruction range and the exception type is the same type or a subclass of the class of exception that the catch clause handles. If a matching catch clause is found, the system branches to the specified handler. If no handler is found, the process is repeated until all nested catch clauses of the current method are exhausted. Catch clauses are ordered and Java.TM. Virtual Machine execution continues at the first matching catch clause. Java.TM. code is structured, therefore all exception handlers for one method may be arranged in a single list. For any possible program counter value, the list can be searched to determine the innermost exception handler that contains the program counter and handles the exception that is thrown.

Detailed Description Text (107):
If no matching catch clause exists for a current method, then the current method has an "uncaught exception" and the execution state of the invoking method is restored. Propagation of the exception continues as though the exception had occurred in the invoker at the instruction that invoked the method within which the exception originated.

Detailed Description Text (108):

The Java.TM. Virtual Machine creates and manipulates objects, including class instances and arrays, using specific instructions for object creation and manipulation. A new instruction creates a new class instance. Newarray, anewarray, and multianewarray instructions create a new array. Getfield, putfield, getstatic, and putstatic instructions access fields of classes (static fields, known as class variables) and fields of class instances (nonstatic fields, known as instance variables). Baload, caload, saload, iaload, laload,faload, daload, and aaload instructions load an array component onto the operand stack. Bastore, castore, sastore, iastore, lastore, fastore, dastore, and aastore instructions store a value from the operand stack as an array component. Arraylength gets the length of an array. Instanceof and checkcast check properties of class instances or arrays. The Java.TM. Virtual Machine invokes methods and returns results using specific instructions. An invokevirtual instruction invokes an instance of a method of an object, dispatching on the virtual type of the object. The invokevirtual instruction is the usual method of dispatch. An invokeinterface instruction invokes a method that is implemented by an interface and searches the methods implemented by the particular runtime object to find the appropriate method. An invokespecial instruction invokes an instance method using special handling, either an instance initialization method <int>, a private method, or a superclass method. An invokestatic instruction invokes a class (static) method in a named class. Method return instructions are distinguished by return type and include ireturn, lreturn, freturn, dreturn, and areturn instructions. A return instruction returns from methods declared as void.

Detailed Description Text (109):
The Java.TM. Virtual Machine allows programming of a thrown exception using an athrow instruction. Exceptions are also thrown by various Java.TM. Virtual Machine instructions upon detection of an abnormal condition.

Detailed Description Text (110):
The new instruction is a bytecode instruction which is called to create a new object. Execution of the new instruction is directed by a class structure including a constant pool. The constant pool includes an objectHint field. The new instruction uses information in the objectHint to create an object structure based on a structure set forth by the constant pool. A single object structure in the constant pool is used to create or instantiate a plurality of objects. The particular class used in the new instruction to instantiate objects is determined by arguments to the instruction. Arguments of the new instruction include an indexbyte1 and an indexbyte2 that are concatenated to form an index into the constant pool of the current class. The item at the specified index to the constant pool must have the tag CONSTANT.sub.-- Class. The symbolic reference is resolved and must result in a class type and not in an array or interface type. Memory for a new instance of the designated class is allocated from a garbage-collecting heap. Instance variables of the new object are initialized to default initial values. The ObjectRef is a reference to the instance and is pushed onto the operand stack. During the resolution of the CONSTANT.sub.-- Class constant pool item, any exception may be thrown. Otherwise, if the CONSTANT.sub.-- Class constant pool item resolves to an interface or is an abstract class, the new instruction throws an InstantiationError. Otherwise, if the current class does not have permission to access the resolved class, the new instruction throws an IllegalAccessError.

Detailed Description Text (111):
The new instruction does not completely create a new instance. Instance creation is not completed until an instance initialization method has been invoked on the uninitialized instance.

Detailed Description Text (112):
Referring to FIG. 8, a schematic block diagram depicts a data structure 800 for implementing an invokevirtual instruction. The invokevirtual instruction is called to invoke an instance method with dispatch based on class. Arguments of the

invokevirtual instruction in an instruction register 802 include an indexbyte1 and an indexbyte2 that are concatenated to form an index or pointer into the constant pool of the current class. The index is held in a constant pool register 804 and applied to address the constant pool 806. The constant pool 806 includes an entry which contains information relating to the method to be invoked by the invokevirtual instruction. The constant pool 806 is an element of the data structure for defining a method to be invoked by an instruction of the instruction set. The constant pool 806 contains an index1 and an index2. The index1is an index to a method reference. The index2 designates a class index which points to a name and type. The class index points to a class structure 808. One entry of the class index designates a name. A second entry designates a type in the form of a string. The class pointer designates the class data structure 808 which designates a method name of the selected method and a method array containing a ByteCode for implementing the method. Upon identification of the ByteCode, the ByteCode is executed. Accordingly, the data structure is used to direct the designated instruction of the instruction set to a class index, to a class pointer, thereby designating a method for execution.

Detailed Description Text (113):
The implementation of hardware follows directly from the specification of an instruction as defined by the Java.TM. Virtual Machine Specification, Lindholm T. and Yellin, F., 1997, Sun Microsystems, Inc. which is hereby incorporated by reference in its entirety. For example, the Java Virtual Machine specification states "if the method is protected, then it must be either a member of the current class or a member of a superclass of the current class". Hardware is implemented to include logic that determines the stated conditions and satisfies the conditions. Thus the hardware accesses the class structure, finds the superclass and the class in the constant pool 806 based on the data structure. The data structure defines the relationship between the memory address and actual data. Each location including a content. The content may be another address, based on the address you search for another content, eventually the data is obtained. The logical operations are implemented using various techniques including digital logic circuits, microcode, trap emulation, and the like.

Detailed Description Text (114):
The item at the specified index to the constant pool must have the tag CONSTANT.sub.-- Methodref, a reference to the class name, a method name, and descriptor of the method. The named method is resolved. The descriptor of the resolved method must be identical to the descriptor of one of the methods of the resolved class. The method must not be either <init>, an instance initialization method, or <clinit>, a class or interface initialization method. If the method is protected, then the method must be either a member of the current class or a member of a superclass of the current class, and the class of objectref must be either the current class or a subclass of the current class.

Detailed Description Text (115):
The constant pool entry representing the resolved method includes an unsigned index into the method table of the resolved class and an unsigned byte nargs that is not equal to zero. The objectref is of the type reference. The index is used as an index into the method table of the class of the type of objectref. If the objectref is an array type, then the method table of the class Object is used. The table entry at that index includes a direct reference to the code for the method and modifier information for the method. The objectref is followed on the operand stack by nargs-1 words of arguments, where the number of words of arguments and the type and order of the values represented is consistent with the descriptor of the selected instance method.

Detailed Description Text (116):
If the method is synchronized, then the monitor associated with the objectref is acquired.

Detailed Description Text (117):
If the method is not native, the nargs-1 words of arguments and objectref are popped from the operand stack. A new stack frame is created for the method invoked, and objectref and the words of arguments are made the values of the first nargs local variables with objectref in local variable 0, arg1 in local variable 1, and the like. The new stack frame is made current and the Java.TM. Virtual Machine program counter (pc) is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

Detailed Description Text (118):
If the method is native and the platform-dependent code that implements the method has not yet been loaded and linked into the Java.TM. Virtual Machine, the method is linked and loaded. The nargs-1 words of arguments and objectref are popped from the operand stack and the code that implements the method is invoked in an implementation-dependent manner.

Detailed Description Text (119):
During resolution of the CONSTANT.sub.-- Methodref constant pool item, any exception can be thrown. Otherwise, if the specified method exists but is a class (static) method, the invokevirtual instruction throws an IncompatibleClassChangeError. Otherwise, if the specified method is abstract and the code that implements the method cannot be loaded or linked, invokevirtual throws and UnsatisfiedLinkError.

Detailed Description Text (121):
Referring to FIG. 9, a schematic block diagram depicts a data structure 900 for implementing an invokestatic instruction. The instructions, including complicated instructions, execute methods that are defined in relation to the class structures. For example, an invokestatic instruction operates as a function call to call other designated functions. Invokestatic is a function call with a plurality of arguments. The arguments are pushed onto an operand stack 901, hardware acquires local variables from a constant pool 902 based on an index defined by the arguments, finds a class that contains a method to be invoked and directs a jump, in the form of a subroutine call to a function called the method. The method is executed. When the method execution is complete, hardware pops the operand stack 901 and executes a return. A decoder (not shown) decodes an instruction and places a decoded code in an instruction register 904 including and instruction designation and an argument designation. The data structure 900 specifies class information such as a class or superclass, an object instantiated from the class, and a method executing an object of the class. The data structure 900 also includes locations in memory that store information for executing the selected method. The data structure 900 includes a constant pool 902 that specifies the class that contains a function and the location of a method to be executed. The processor 202 accesses information in the data structure 900 and operates on the information to execute a selected object method. Once an instruction is decoded as an invokestatic instruction, the next two bytes in the instruction register 904 are defined to specify an index to the constant pool 902. A constant pool register 906 is set according to the decoded instruction and index information to access an entry in the constant pool 902 that contains the invokestatic method. Hardware (not shown) accesses the content of the entry in the constant pool 902 and receives an identifier of another entry in the constant pool 902 that includes a content that identifies the name of a method for execution in addition to a type and a signature of the method. The signature is functional argument of the method in the form of a string.

Detailed Description Text (122):
The name, type and signature information in the constant pool 902 are accessed and used to control the operations of logic that determine the functionality of the object-oriented processor 202. The specific logic follows explicitly according to the defined configuration. In an illustrative embodiment, the defined configuration

is based on the Java Virtual Machine Specification including an instruction set definition that describe precisely how the multiple instructions function. The Java.TM. Virtual Machine Specification describes the content of the constant pool 902 and directs how the content of the constant pool 902 is translated to logical operations. The Class structure supplements the description of the instruction set in the Java.TM. Virtual Machine specification to define the operating logic of the object-oriented processor 202. The logic may take several forms such as digital logic circuitry, microcode, or execution by trap emulation. In one embodiment, logic hardware is used to implement simple instructions such as instructions operable with respect to static classes, for example the invokestatic instruction, while microcode and trap emulation operations are used to implement more complicated instructions. Hardware logic advantageously achieves an improved performance. Microcode and trap emulation operations advantageously simplify complicated logic operations and reduce circuit size.

Detailed Description Text (123):
In some embodiments, a hardware implementation of the instruction set defined by the Java.TM. Virtual Machine Specification is explicitly determined by the specification of each instruction. For example, the invokestatic instruction specifies that unsigned indexbyteland indexbyte2 are used to construct an index into the constant pool 902 where the value of the index is (indexbyte1<<8).vertline. indexbyte2. Hardware logic shifts the indexbyte1 by eight bits, concatenates the shifted indexbyte1 and indexbyte2, and applies the resulting concatenated word as a pointer to the constant pool 902.

Detailed Description Text (124):
The Java.TM. Virtual Machine Specification then states that the item at the index address have the tag CONSTANT.sub.-- MethodRef, a reference to the class name, a method name, and the method's descriptor. This statement specifies a required structure for the constant pool 902. Hardware includes logic for resolving the constant pool 902 structure, determining whether the required structure is implemented in the constant pool 902 and activating an exception for violations. The Java.TM. Virtual Machine Specification designates that the named method is resolved, dynamically determining concrete values from symbolic references in the constant pool 902. To resolve the method reference, hardware logic resolves the CONSTANT.sub.-- Class entry representing the class of which the method is a member and throws an exception if resolution is not successfully achieved. The name.sub.-- index of a CONSTANT.sub.-- Class constant pool entry is a reference to a CONSTANT.sub.-- Utf8 constant pool for a UTF-8 string that represents a fully qualified name of the class to be resolved. Hardware logic determines the type of entry that is represented by the CONSTANT.sub.-- Class constant pool entry and resolves the entry as follows. First, hardware logic includes a comparator for determining whether the first character of the fully qualified name of the constant pool entry to be resolved is a left bracket ("["). The entry is an array class if the entry is a left bracket and otherwise is in a nonarray class or to an interface. Hardware logic then directs operations in alternate paths depending on the type of class. If the first character of the fully qualified name of the constant pool entry to be resolved is a left bracket, then the entry is a reference to an array class and a special resolution is invoked.

Detailed Description Text (125):
For the non-array class, hardware logic first loads the designated class and associated superclasses by first finding, then loading, the class and superclasses. If no file with an appropriate name is found and read, hardware logic throws a NoClassDefFoundError. Otherwise, if the hardware logic tests characteristics of the selected file and determines that the file is not a well-formed class file or is not a class file of a supported major or minor version, class or interface, then the hardware throws a NoClassDefFoundError. The NoClassDefFoundError is also thrown by the hardware logic if the hardware logic determines that the selected class file does not contain the desired class or interface. If the selected class file does

not specify a superclass and is not the class file for the class file for class Object, then the class or interface resolution throws a ClassFormatError.

Detailed Description Text (126):
If the superclass of the specified class to be loaded is not yet loaded, hardware detects any loading errors, then recursively loads the class. All interfaces are to have a previously loaded Java.lang.Object as a superclass.

Detailed Description Text (127):
If loading of the class and superclass was successful, the hardware logic links and initializes the class. Hardware links a class by verifying that the class is suitable and preparing the class for execution. First, the hardware logic verifies the class or interface to ensure that the binary representation of the class is structurally valid. Verification may cause classes and interfaces to be loaded but not initialized. If the class or interface contained in a class file does not satisfy static or structural constraints imposed on valid class files, the hardware logic throws a VerifyError. If the hardware logic successfully verifies the class or interface, then hardware logic prepares the class or interface. Hardware logic prepares the class or interface by creating static fields for the class or interface and initializing the fields to standard default values. During preparation, hardware logic throws an AbstractMethodError if the class is not declared abstract and has an abstract method.

Detailed Description Text (128):
Hardware logic then initializes the class. If hardware logic serving as an initializer completes abruptly by throwing an exception and the class of the exception is not the class Error or one of the subclasses of the class Error, then the hardware logic creates an instance of the class ExceptionInInitializerError which is used in place of the Error object. If the hardware logic unsuccessfully attempts to create a new instance of the class ExceptionInInitializationError because an OutOfMemoryError occurs, then the hardware logic throws the OutOfMemoryError.

Detailed Description Text (129):
Hardware logic then checks the access permissions of the class being resolved. If the current class or interface does not have permission to access the class or interface being resolved, then the hardware logic throws an IllegalAccessError.

Detailed Description Text (130):
If the hardware logic detects no errors, constant pool resolution of the class or interface is successful. Alternatively, if the hardware logic detects an error, resolution is unsuccessful and the hardware logic prevents accessing of the referenced class or interface.

Detailed Description Text (131):
If the constant pool resolution of the class or interface is successful, hardware logic then performs further tests to determine whether linking of the method or field is successfully resolved. Hardware logic determines whether the referenced field exists in the specified class or interface and, if not, throws a NoSuchFieldError. The hardware logic determines whether the current class has permission to access the referenced field and, if not, throws an IllegalAccessError exception.

Detailed Description Text (132):
The hardware logic compares the descriptor of the resolved method and verifies that the descriptor is identical to the descriptor of one of the methods of the resolved class. The hardware logic verifies that the method is neither an instance initialization method (<init>) nor a class or interface initialization method (<clinit>). The hardware logic verifies that the method is static and not abstract. The hardware logic determines whether the method is protected and, if so, verifies

that the method is either a member of the current class or a member of a superclass of the current class.

Detailed Description Text (133):
The entry constant pool 902 representing the resolved method includes a direct reference to the code for the method, an unsigned byte nargs that may be zero, and the modifier information of the method.

Detailed Description Text (134):
The hardware logic verifies that the operand stack 901 contains nargs words of arguments, where the number nargs and the type and order of the values of the arguments is consistent with the descriptor of the resolved method.

Detailed Description Text (135):
For a method that is synchronized, the hardware logic acquires the monitor associated with the current class.

Detailed Description Text (136):
For a method that is not "native", hardware logic pops the nargs words from the operand stack 901. The hardware logic creates a new stack frame for the method invoked and transfers the words of the arguments to the first nargs local variables. The hardware logic makes the new stack frame current and sets a program counter 904 to the opcode of the first instruction of the method to be invoked and execution continues with the first instruction of the method.

Detailed Description Text (137):
If the method is "native", the hardware logic pops the nargs words of arguments from the operand stack 901. The hardware logic then invokes the method.

Detailed Description Text (138):
The illustrative object-oriented processor 202 is a hardware implementation of instructions defined by the Java.TM. Virtual Machine. Hardware implements the instructions of the Java instruction set with a data structure including an object structure defining the structure and operations of the hardware. The hardware is typically and variously implemented in different embodiments using digital hardware logic, microcode, and trap emulation. In other embodiments, other known technologies such as state machines, analog logic, and the like may be utilized. In the illustrative embodiment of an object-oriented processor 202, the instructions of the instruction set are implemented as a method or algorithm in the form of a computer operation operating with respect to a data structure. In contrast, a conventional Java.TM. Virtual Machine processor operates a software application at a level higher than a software operating system.

Detailed Description Text (139):
Instructions of the instruction set include hardware that generally operates in the manner of the hardware for implementing the illustrative invokestatic instruction including accessing a constant pool, finding a class and then a method based on the information in the constant pool, then executing the method. The memory of the constant pool is consistently laid out so that a two-byte index is sufficient for designating a particular method. More simple instructions generally use a smaller portion of the data structure to define the operation of the method. Some instructions simply use the operand stack alone. More complex instructions generally use a larger portion of the data structure using indirect memory accessing to traverse several levels of the data structure, acquiring additional information for executing the method in the multiple levels of the data structure.

Detailed Description Text (140):
The putstatic instruction is called to set a static field in a class. Arguments of the putstatic instruction include an indexbyte1 and an indexbyte2 that are concatenated to form an index into the constant pool of the current class. The item

at the specified index to the constant pool must have the tag CONSTANT.sub.-- Methodref, a reference to the class name, and a field name. If the method is protected, then the method must be either a member of the current class or a member of a superclass of the current class.

Detailed Description Text (141):
The constant pool item is resolved, determining both the class field and the class field width. The type of value stored by a putstatic instruction must be compatible with the descriptor of the field of the class instance into which the value is stored. The value is popped from the operand stack and the class field is set to value.

Detailed Description Text (142):
During resolution of the CONSTANT.sub.-- Methodref constant pool item, any exception can be thrown. Otherwise, if the specified field exists but is not a class static field (a class variable), the putstatic instruction throws an IncompatibleClassChangeError.

Detailed Description Text (143):
The athrow instruction throws an exception or error. The athrow does not include an argument. An objectref must be of the type reference and refer to an object which is an instance of class Throwable or of a subclass of Throwable. The objectref is popped from the operand stack. The objectref is then thrown by searching the current frame for the most recent catch clause that catches the class of objectref or one of the superclasses of the objectref. It a catch clause is found, the catch clause contains the location of the code intended to handle the exception. The pc register is reset to the catch clause location, the operand stack of the current frame is cleared, objectref is pushed back onto the operand stack, and execution continues. If no appropriate clause is found in the current frame, the frame is popped, and the frame of the method invoker is reinstated, and the objectref is rethrown. If no catch clause is found that handles this exception, the current thread exits.

Detailed Description Text (145):
The operand stack operation for the athrow instruction functions as follows: If a handler for an exception is found in the current method, the athrow instruction discards all words on the operand stack, then pushes the thrown object onto the stack. However, if no handler is found in the current method and the exception is thrown farther up the method invocation chain, then the operand stack of the method, if any, that handles the exception is cleared and objectref is pushed onto the empty operand stack. All intervening stack frames from the method that threw the exception up to but not including the method that handles the exception are discarded.

Detailed Description Text (146):
While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Many variations, modifications, additions and improvements of the embodiments described are possible. For example, those skilled in the art will readily implement the steps necessary to provide the structures and methods disclosed herein, and will understand that the described parameters, materials, and dimensions are given by way of example only and can be varied to achieve the desired structure as well as modifications which are within the scope of the invention. Variations and modifications of the embodiments disclosed herein may be made based on the description set forth herein, without departing from the scope and spirit of the invention as set forth in the following claims.

CLAIMS:

1. An object-oriented processor comprising:

a memory storing a data structure including a class structure defined on an object-oriented basis;

an execution unit coupled to the memory including a logic for deriving a pointer into the class structure, the pointer designating a class for instantiating an object and executing the object as a method; and

a plurality of registers for directing the execution of object methods according to a predetermined data structure design the plurality of registers including:

an instruction register;

a constant pool register:

a frame register; and

a class register.

3. An object-oriented processor according to claim 1, wherein:

the data structure in the memory is a multiple-level data structure; and

the execution unit includes a logic for deriving a plurality of pointers to the class structure for accessing a plurality of locations in the memory to index a plurality of locations in multiple-part instruction.

4. An object-oriented processor according to claim 1, wherein:

an instruction set for the object-oriented processor is defined by a Java.TM. Virtual Machine specification.

5. An object-oriented processor according to claim 1, wherein:

the memory further includes a constant pool storing condensed information of the class structure defining multiple-level memory accesses to index parts of a multiple-part instruction.

6. A method of implementing an object-oriented processor comprising:

defining a data structure of an object-oriented operating environment;

defining an object-oriented instruction set for executing in the object-oriented operating environment;

directing the execution of object methods with a plurality of registers according to a predetermined data structure design, the plurality of registers including:

an instruction register;

a constant pool register;

a frame register; and

a class register; and

generating a hardware implementation of the processor enforcing the logical relationships of the instruction set as defined by the data structure.

7. A method according to claim 6 wherein the hardware implementation generating

operation is selected from among a group consisting of:

trap emulation;

a microcode sequence; and

a hard-coded digital hardware logic.

8. A method according to claim 6 wherein the object-oriented instruction set is defined by a Java.TM. Virtual Machine specification.

9. A method according to claim 6 wherein the data structure includes a class structure and an object structure.

12. An object-oriented processor according to claim 10 wherein the object-oriented instruction set is defined by a Java.TM. Virtual Machine specification.

13. An object-oriented processor according to claim 10 wherein the data structure is a memory including a class structure and an object structure.

14. An object-oriented processor comprising:

a plurality of registers for directing the execution of object methods according to a predetermined data structure design, the plurality of registers including:

an instruction register;

a constant pool register;

a frame register; and

a class register;

a memory including data structures that are defined for objects that are to be executed by the processor, the memory implementing an object-oriented instruction set in accordance with the data structures; and

an execution unit for executing object methods according to the object-oriented instruction set implemented in the memory.

15. An object-oriented processor according to claim 14 wherein the data structure design includes:

a constant pool including an entry having a pointer to a class structure;

the class structure having a pointer to an object structure; and

the object structure having a pointer to a method.

17. An object-oriented processor according to claim 14 wherein the object-oriented instruction set is defined by a Java.TM. Virtual Machine specification.

18. An object-oriented processor according to claim 14 wherein the data structure is a memory including a class structure and an object structure.

argument and the stack pointer (sp) is adjusted to make room for local variables to be stored on the stack.

Drawing Description Text (7):
FIG. 6 illustrates the internal organization of the local-variable region of the stack activation frame. This region includes application-declared locals (as declared in byte-code attributes for Java methods and as specified in the parameterization of BuildFrames( ), temporary variables (as might be required to represent the old values of the frame and instruction pointers), a run-time stack (to allow execution of push and pop operations within the method), and space for arguments to be pushed to other methods to be called from this method.

Drawing Description Text (14):
FIG. 13 provides C preprocessor definitions of symbolic constants used to describe the encodings of Sun's Java byte code instruction set.

Drawing Description Text (18):
FIG. 17 provides the C declaration of the structure used internal to the PERC implementation to represent a raw class file that has been read into memory. The class-file loader analyzes this object to create an appropriate Class representation.

Drawing Description Text (29):
FIG. 28 provides a C macro definition of the SetJmp( ) macro, which is a version of the standard C setjmp( ) function specialized for the PERC virtual machine execution environment.

Drawing Description Text (30):
FIG. 29 provides a C macro definition of the UnsetJmp( ) macro, which is used within the PERC virtual machine execution environment to replace the current exception handling context with the surrounding exception handling context.

Drawing Description Text (31):
FIG. 30 provides a C macro definition of the LongJmpo macro, which is a version of the standard C longjmp( ) function specialized for the PERC virtual machine execution environment. Note that this macro makes use of perclongjmp( ) whose implementation is not provided. perclongjmp( ) expects as parameters a representation of the machine's registers including its instruction pointer, the value of the pointer stack pointer, the value of the non-pointer stack pointer, and the return value to be returned to the point of the JIT version of setjmp( ).

Drawing Description Text (54):
FIG. 53 provides the Java implemenation of the TaskDispatcher class .

Drawing Description Text (67):
FIG. 66 provides the definition of a C macro used within the implementation of the PERC virtual machine to support preemption of the currently executing thread.

Drawing Description Text (68):
FIG. 67 provides the definitions of C macros for saving and restoring the state of the PERC virtual machine surrounding each preemption point.

Drawing Description Text (69):
FIG. 68 provides the C implementation of the PERC virtual machine, except that cases to handle each byte code are excluded.

Drawing Description Text (70):
FIG. 69 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the IADD instruction, which adds the two integers on the top of the Java stack, placing the result on the Java stack.

Drawing Description Text (71):
FIG. 70 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the AASTORE instruction.

Drawing Description Text (72):
FIG. 71 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the FCMPL instruction.

Drawing Description Text (73):
FIG. 72 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the IFEQ instruction.

Drawing Description Text (74):
FIG. 73 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the JSR instruction.

Drawing Description Text (75):
FIG. 74 provides the C code to be inserted into the PERC virtual machine

Drawing Description Text (77):
FIG. 75 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the TABLESWITCH instruction.

Drawing Description Text (78):
FIG. 76 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the LOOKUPSWITCH instruction.

Drawing Description Text (79):
FIG. 77 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the IRETURN instruction.

Drawing Description Text (80):
FIG. 78 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the GETSTATIC.sub.-- QNP8 instruction.

Drawing Description Text (81):
FIG. 79 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the PUTFIELD.sub.-- Q instruction.

Drawing Description Text (82):
FIG. 80 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKEVIRTUAL.sub.-- FQ instruction.

Drawing Description Text (83):
FIG. 81 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKESPECIAL.sub.-- Q instruction.

Drawing Description Text (84):
FIG. 82 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKESTATIC.sub.-- Q instruction.

Drawing Description Text (85):
FIG. 83 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKEINTERFACE.sub.-- Q instruction.

Drawing Description Text (86):
FIG. 84 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the NEW.sub.-- Q instruction.

Drawing Description Text (87):
FIG. 85 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the NEWARRAY instruction. FIG. 86 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the ANEWARRAY.sub.-- Q instruction.

Drawing Description Text (88):
FIG. 87 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the ATHROW instruction.

Drawing Description Text (89):
FIG. 88 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the CHECKCAST.sub.-- Q instruction.

Drawing Description Text (90):
FIG. 89 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INSTANCEOF.sub.-- Q instruction.

Drawing Description Text (91):
FIG. 90 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the MONITORENTER instruction.

Drawing Description Text (92):
FIG. 91 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the MONITOREXIT instruction.

Drawing Description Text (97):
FIG. 96 illustrates the Java implementation of the Atomic class for use on uniprocessor systems that lack the capability to analyze worst-case execution times. Application programmers can prevent threads from being preempted within certain critical regions by surrounding those regions with execution of Atomic.enter( ) and Atomic.exit( ).

Detailed Description Text (3):
The PERC virtual machine consists primarily of an interpreter for the PERC byte-code instruction set, a task (thread) dispatcher, and a garbage collector written in C which runs as an independent real-time task. Most of the functionality of the PERC execution environment is provided by standard library and system programs that accompany the virtual machine and are executed by the virtual machine.

Detailed Description Text (4):
PERC (and Java) is an object-oriented programming language. Programs are comprised of object type declarations, known in PERC as classes. Each class definition describes the variables that are associated with each instance (object) of the corresponding class and also defines all of the operations that can be applied to instantiated objects of this type. Operations are known as methods.

Detailed Description Text (12):
Methods represented as byte codes are interpreted by the PERC virtual machine. The interpreter, known throughout this invention disclosure as pvm() (for PERC virtual machine), uses three stacks for execution: (1) the traditional C stack, (2) an explicitly managed stack for representation of PERC pointer values, and (3) an explicitly managed stack for representation of PERC non-pointer values. The C stack holds C-declared local variables and run-time state information associated with compiler generated temporaries. The PERC pointer stack holds the pointer arguments

passed as inputs to the method, pointer local variables, temporary pointers pushed during expression evaluation, and pointer values pushed as arguments to methods called by the current method. The PERC non-pointer stack holds non-pointer arguments passed as inputs to the method, non-pointer local variables, temporary non-pointer values pushed during expression evaluation, and non-pointer values pushed as arguments to be called by this method. The pointer and non-pointer stack activation frames are illustrated in FIG. 5 and FIG. 6.

Detailed Description Text (16):
Native methods use the same three stacks as are used by the PERC virtual machine to execute byte-code methods.

Detailed Description Text (18):
PERC, like Java, supports four distinct forms of method invocation. These are known as (1) virtual, (2) special (non-virtual), (3) static, and (4) interface. With virtual and special method invocations, there is an implicit (not seen by the Java programmer) "this" argument passed to the called method. The "this" argument refers to the object on which the called method will operate. The distinctions between these different method invocations are described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley.

Detailed Description Text (20):
The PERC implementation represents every PERC object with a data structure patterned after the templates provided in FIG. 15, FIG. 16, and FIG. 24. In all of these structures, the second field is a pointer to a MethodTable data structure (see FIG. 23). The PERC execution environment maintains one MethodTable data structure for each defined object type. All instantiated objects of this type point to this shared single copy. The jit.sub.-- interfaces array field of the MethodTable structure has one entry for each virtual method supported by objects of this type. The mapping from method name and signature to index position is defined by the class loader, as described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley. To execute the JIT version of a PERC method using a virtual method lookup, branch to the code represented by jit.sub.-- interfaces[method.sub.-- index]. Normally, the JIT version of the byte code will only be invoked directly from within another JIT-compiled method. If a native or untranslated byte-code method desires to invoke another method using virtual method lookup, the search for the target method generally proceeds differently. First, we find the target object's MethodTable data structure (as above) and then follow the methods pointer to obtain an array of pointers to Method objects. Within the Method object, we consult the access.sub.-- flags field to determine if the target method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be implemented by byte codes. See FIG. 49, FIG. 45, and FIG. 46.

Detailed Description Text (22):
When the method to be invoked by a particular operation is known at compile time, the Java compiler treats this as an invokeSpecial instruction. In these cases, there is no need to consult the method table at run time. When performing special method invocation from within a JIT-translated method, the address of the called method (or at least a stub for the called method) is hard-coded into the caller's machine code.

Detailed Description Text (25):
When the method to be invoked is declared as static within the corresponding object (meaning that the method operates on class information rather than manipulating variables associated with a particular instance of the corresponding class), the Java compiler treats this as an invokeStatic method. Execution of static methods is identical to execution of special methods except that there is no implicit pointer to "this" passed as an argument to the called method. See FIG. 47, FIG. 45, and FIG. 46.

```
@PJL JOB NAME="http://westbrs:9000/bin/gate.exe?f=doc&state=sf5tlh.34.1&ESNAME"
@PJL SET RET=ON
@PJL SET DUPLEX=OFF
@PJL SET ECONOMODE=OFF
@PJL SET OUTBIN=UPPER
@PJL SET FINISH=NONE
@PJL SET PAGEPROTECT=AUTO
@PJL SET PAPER=LETTER
@PJL SET HOLD=OFF
@PJL SET RESOLUTION=600
@PJL ENTER LANGUAGE=PCL
```

First Hit Fwd Refs
End of Result Set

☐ Generate Collection Print

L21: Entry 1 of 1 File: USPT Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A
TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Abstract Text (1):
The invention is a method for use in executing portable virtual machine computer programs under real-time constraints. The invention includes a method for implementing a single abstract virtual machine execution stack with multiple independent stacks in order to improve the efficiency of distinguishing memory pointers from non-pointers. Further, the invention includes a method for rewriting certain of the virtual machine instructions into a new instruction set that more efficiently manipulates the multiple stacks. Additionally, using the multiple-stack technique to identify pointers on the run-time stack, the invention includes a method for performing efficient defragmenting real-time garbage collection using a mostly stationary technique. The invention also includes a method for efficiently mixing a combination of byte-code, native, and JIT-translated methods in the implementation of a particular task, where byte-code methods are represented in the instruction set of the virtual machine, native methods are written in a language like C and represented by native machine code, and JIT-translated methods result from automatic translation of byte-code methods into the native machine code of the host machine. Also included in the invention is a method to implement a real-time task dispatcher that supports arbitrary numbers of real-time task priorities given an underlying real-time operating system that supports at least three task priority levels. Finally, the invention includes a method to analyze and preconfigure virtual memory programs so that they can be stored in ROM memory prior to program.

Brief Summary Text (4):
Java (a trademark of Sun Microsystems, Inc.) is an object-oriented programming language with syntax derived from C and C++. However, Java's designers chose not to pursue full compatibility with C and C++ because they preferred to eliminate from these languages what they considered to be troublesome features. In particular, Java does not support enumerated constants, pointer arithmetic, traditional functions, structures and unions, multiple inheritance, goto statements, operator overloading, and preprocessor directives. In their place, Java requires all constant identifiers, functions (methods), and structures to be encapsulated within

Brief Summary Text (6):
One distinguishing characteristic of Java is its execution model. Java programs are first translated into a fully portable standard byte code representation. The byte code is then available for execution on any Java virtual machine. A Java virtual machine is simply any software system that is capable of understanding and executing the standard Java byte code representation. Java virtual machine support is currently available for AIX, Apple Macintosh, HPUX, Linux, Microsoft NT, Microsoft Windows 3.1, Microsoft Windows 95, MVS, Silicon Graphics IRIX, and Sun Solaris. Ports to other environments are currently in progress. To prevent viruses from being introduced into a computer by a foreign Java byte-code program, the Java virtual machine includes a Java byte code analyzer that verifies the byte code does

not contain requests that would compromise the local system. By convention, this byte code analyzer is applied to every Java program before it is executed. Byte code analysis is combined with optional run-time restrictions on access to the local file system for even greater security. Current Java implementations use interpreters to execute the byte codes but future high-performance Java systems will have the capability of translating byte codes to native machine code on the fly. In theory, this will allow Java programs to run approximately at the same speed as C++.

Brief Summary Text (7):
Within Sun, development of Java began in April of 1991. Initially, Java was intended to be an implementation language for personal digital assistants. Subsequently, the development effort was retargeted to the needs of set-top boxes, CD-ROM software, and ultimately the World-Wide Web. Most of Java's recent media attention has focused on its use as a medium for portable distribution of software over the Internet. However, both within and outside of Sun, it is well understood that Java is much more than simply a language for adding animations to Web pages. In many embedded real-time applications, for example, the Java byte codes might be represented in system ROMs or might even be pre-translated into native machine code.

Brief Summary Text (10):
This invention relates generally to computer programming methods pertaining to real-time applications and more specifically to programming language implementation methods which enable development of real-time software that can run on computer systems of different designs. PERC (a trademark of NewMonics Inc.) is a dialect of the Java programming language designed to address the special needs of developers of real-time software.

Brief Summary Text (15):
Byte code is a term of art that describes a method of encoding instructions (for interpretation by a virtual machine) as 8-bit numbers, each pattern of 8 bits representing a different instruction.

Brief Summary Text (27):
JIT, as the term is used in this invention disclosure, is an acronym standing for "just in time." The term is used to describe a system for translating Java byte codes to native machine language on the fly, just-in-time for its execution. We consider any translation of byte code to machine language which is carried out by the virtual machine to be a form of JIT compilation.

Brief Summary Text (29):
Method is a term of art describing the unit of procedural abstraction in an object-oriented programming system. All methods are associated with particular class definitions. Rather than calling a procedure or function, the object-oriented programmer invokes the method associated with the data object on which the method is intended to operate.

Brief Summary Text (30):
Native Method, as this term is used in relation to the Java and PERC programming languages, describes a method that is implemented in C (or some other low-level language) rather than in the high-level Java or PERC language in which the majority of methods are implemented.

Brief Summary Text (37):
RTVMM, as the term is used in this invention disclosure, is an acronym standing for Real-Time Virtual Machine Method. This acronym represents the invention disclosed by this document.

Brief Summary Text (43):

Thread is a term of art describing a computer program that executes with an independent flow of execution. Java is a threaded language, meaning that multiple flows of execution may be active concurrently. All threads share access to the same global memory pool. (In other programming environments, threads are known as tasks.)

Brief Summary Text (47):
The invention is a real-time virtual machine method (RTVMM) for use in implementing portable real-time systems. The RTVMM provides efficient support for execution of portable byte-code representations of computer programs, including support for accurate defragmenting real-time garbage collection. Efficiency is measured both in terms of memory utilization, CPU time, and programmer productivity. Programmer productivity is enhanced through reduction of the human effort required to make the RTVMM available in multiple execution environments.

Brief Summary Text (50):
2. A mechanism to translate traditional Java byte codes into the extended PERC byte codes at run-time, as new Java byte codes are loaded into the virtual machine's execution environment.

Brief Summary Text (52):
of the PERC instruction set. The Java run-time stack is replaced by two stacks, one for non-pointer and the other for pointer data. Further, the data structures enable efficient interaction between native methods, Java methods represented by byte code, and Java methods translated by a JIT compiler to native machine language. Performance tradeoffs are biased to give favorable treatment to execution of JIT-translated methods.

Brief Summary Text (53):
4. A set of C macros and functions that characterize the native-method application programmer interface (API). This API abstracts the native-method programmer's interface to the internal data structures, the run-time task scheduler, and the garbage collector.

Brief Summary Text (54):
5. A method for implementing mostly stationary defragmenting real-time garbage collection in software.

Brief Summary Text (55):
6. A method for supporting arbitrary numbers of task priority levels and control over dispatching of individual tasks using an underlying operating system that provides fixed priority preemptive scheduling with a minimum of three priority levels.

Brief Summary Text (56):
7. A mechanism for translating traditional Java byte codes into the extended PERC byte codes prior to run-time, in order to reduce run-time overhead and simplify system organization.

Drawing Description Text (6):
FIG. 5 illustrates the appearance of the pointer and non-pointer stack activation frames immediately before calling and immediately following entry into the body of a Java method. The stacks are assumed to grow downward. In preparation for the call, arguments are pushed onto the stack. Within the called method, the frame pointer (fp) is adjusted to point at the memory immediately above the first pushed argument and the stack pointer (sp) is adjusted to make room for local variables to be stored on the stack.

Drawing Description Text (7):
FIG. 6 illustrates the internal organization of the local-variable region of the

stack activation frame. This region includes application-declared locals (as declared in byte-code attributes for Java methods and as specified in the parameterization of BuildFrames( ), temporary variables (as might be required to represent the old values of the frame and instruction pointers), a run-time stack (to allow execution of push and pop operations within the method), and space for arguments to be pushed to other methods to be called from this method.

Drawing Description Text (13):
FIG. 12 provides C preprocessor definitions of symbolic constants used to describe access flags in the representations of classes, methods, and fields.

Drawing Description Text (18):
FIG. 17 provides the C declaration of the structure used internal to the PERC implementation to represent a raw class file that has been read into memory. The class-file loader analyzes this object to create an appropriate Class representation.

Drawing Description Text (19):
FIG. 18 provides the C declaration of the structure used internal to the PERC implementation to represent the range of byte code instructions within a byte-code method to which a particular exception handler applies. The handler.sub.-- pc field is the offset of the exception handler code.

Drawing Description Text (20):
FIG. 19 provides the C declaration of the structure used internal to the PERC implementation to represent a field within a class. The constantvalue.sub.-- index field is used during loading to represent the offset within the constant pool of the value of each static final field. After the offsets of each field within the class's static data region have been determined (as represented by the Field structure's offset field), the constant is copied out of the constant pool into the corresponding data location.

Drawing Description Text (23):
FIG. 22 provides the C declaration of the structure used internal to the PERC implementation to represent a Method structure.

Drawing Description Text (41):
FIG. 40 provides C macros for conversion between integer offsets and actual derived pointer values and for obtaining the actual address of the constant-pool object. These macros are used to improve the efficiency of access to instruction, stack, and constant-pool memory.

Drawing Description Text (46):
FIG. 45 provides C macros for use by C code invocations of PERC methods.

Drawing Description Text (55):
FIG. 54 provides the C implementation of the TaskDispatcher's critical native methods and help routines.

Drawing Description Text (64):
FIG. 63 provides the definitions of C macros for use in returning values from native methods and C helper functions.

Drawing Description Text (95):
FIG. 94 illustrates the PERC non-pointer stack activation frame for JIT-generated code. Upon entry into the JIT function, the non-pointer stack pointer (npsp) points to the list of incoming arguments, and the return address is stored in the slot "above" the top-of-stack entry. The prologue of JIT-compiled method subtracts a JIT-computed constant from npsp to make room on the non-pointer stack for saved machine registers, local variables, and outgoing arguments.

Drawing Description Text (96):
FIG. 95 illustrates the organization of free lists, partitioned by region, but combined into a single global pool to support efficient constant-time allocation. In this figure, the three regions (indicated by the three large objects on the left side of the figure) are prioritized such that preference is given to allocating from the top region first followed by the middle region and then the bottom region. This figure illustrates only two size categories, 16 and 32. In the actual implementation, there are free lists for each size category, ranging from size 4 to size 512K.

Drawing Description Text (98):
FIG. 97 illustrates the native-method implementations of the Atomic.enter( ) and Atomic.exit( ) methods, respectively.

Detailed Description Text (4):
PERC (and Java) is an object-oriented programming language. Programs are comprised of object type declarations, known in PERC as classes. Each class definition describes the variables that are associated with each instance (object) of the corresponding class and also defines all of the operations that can be applied to instantiated objects of this type. Operations are known as methods.

Detailed Description Text (5):
Internally, PERC methods are represented using one of three different forms:

Detailed Description Text (6):
1. The PERC programmer can choose to implement certain methods in C. At run-time, these methods are represented by native machine code. Such methods are known as native methods.

Detailed Description Text (7):
2. All other PERC methods are written in PERC. At run time, certain methods written in PERC are represented as PERC byte codes.

Detailed Description Text (8):
3. The PERC-written methods that are not represented as PERC byte codes have been translated to native machine language by a JIT compiler.

Detailed Description Text (9):
2.0 Execution Modes for PERC Methods

Detailed Description Text (10):
There are three different modes of execution for PERC methods. Special effort is required to switch between these execution modes since they use the run-time stack (s) differently.

Detailed Description Text (11):
2.1 Byte-code methods

Detailed Description Text (12):
Methods represented as byte codes are interpreted by the PERC virtual machine. The interpreter, known throughout this invention disclosure as pvm() (for PERC virtual machine), uses three stacks for execution: (1) the traditional C stack, (2) an explicitly managed stack for representation of PERC pointer values, and (3) an explicitly managed stack for representation of PERC non-pointer values. The C stack holds C-declared local variables and run-time state information associated with compiler generated temporaries. The PERC pointer stack holds the pointer arguments passed as inputs to the method, pointer local variables, temporary pointers pushed during expression evaluation, and pointer values pushed as arguments to methods called by the current method. The PERC non-pointer stack holds non-pointer

arguments passed as inputs to the method, non-pointer local variables, temporary non-pointer values pushed during expression evaluation, and non-pointer values pushed as arguments to be called by this method. The pointer and non-pointer stack activation frames are illustrated in FIG. 5 and FIG. 6.

Detailed Description Text (13):
2.2 JIT-compiled methods

Detailed Description Text (14):
Methods that have been translated to native machine code use only two stacks: the PERC pointer stack and the PERC non-pointer stack. The benefit of using only two rather than three stacks is that this reduces the overhead of stack maintenance associated with each method invocation. The activation frames for the two stacks are structured as illustrated in FIG. 94. However, the amount of information stored in the "temporaries" segment of the activation frame differs between JIT-compiled methods and byte-code methods.

Detailed Description Text (15):
2.3 Native methods

Detailed Description Text (16):
Native methods use the same three stacks as are used by the PERC virtual machine to execute byte-code methods.

Detailed Description Text (17):
3.0 Method Invocation

Detailed Description Text (18):
PERC, like Java, supports four distinct forms of method invocation. These are known as (1) virtual, (2) special (non-virtual), (3) static, and (4) interface. With virtual and special method invocations, there is an implicit (not seen by the Java programmer) "this" argument passed to the called method. The "this" argument refers to the object on which the called method will operate. The distinctions between these different method invocations are described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley.

Detailed Description Text (19):
3.1 Virtual Invocation of Methods

Detailed Description Text (20):
The PERC implementation represents every PERC object with a data structure patterned after the templates provided in FIG. 15, FIG. 16, and FIG. 24. In all of these structures, the second field is a pointer to a MethodTable data structure (see FIG. 23). The PERC execution environment maintains one MethodTable data structure for each defined object type. All instantiated objects of this type point to this shared single copy. The jit.sub.-- interfaces array field of the MethodTable structure has one entry for each virtual method supported by objects of this type. The mapping from method name and signature to index position is defined by the class loader, as described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley. To execute the JIT version of a PERC method using a virtual method lookup, branch to the code represented by jit.sub.-- interfaces[method.sub.-- index]. Normally, the JIT version of the byte code will only be invoked directly from within another JIT-compiled method. If a native or untranslated byte-code method desires to invoke another method using virtual method lookup, the search for the target method generally proceeds differently. First, we find the target object's MethodTable data structure (as above) and then follow the methods pointer to obtain an array of pointers to Method objects. Within the Method object, we consult the access.sub.-- flags field to determine if the target method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be

implemented by byte codes. See FIG. 49, FIG. 45, and FIG. 46.

Detailed Description Text (21):
3.2 Special Invocation of Methods

Detailed Description Text (22):
When the method to be invoked by a particular operation is known at compile time, the Java compiler treats this as an invokeSpecial instruction. In these cases, there is no need to consult the method table at run time. When performing special method invocation from within a JIT-translated method, the address of the called method (or at least a stub for the called method) is hard-coded into the caller's machine code.

Detailed Description Text (23):
If a native or untranslated byte-code method desires to perform the equivalent of an invokeSpecial operation, we examine the Method object that represents the target procedure and consult its access.sub.-- flags field to determine if the method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be represented as byte code. See FIG. 48, FIG. 45, and FIG. 46.

Detailed Description Text (24):
3.3 Static Invocation of Methods

Detailed Description Text (25):
When the method to be invoked is declared as static within the corresponding object (meaning that the method operates on class information rather than manipulating variables associated with a particular instance of the corresponding class), the Java compiler treats this as an invokeStatic method. Execution of static methods is identical to execution of special methods except that there is no implicit pointer to "this" passed as an argument to the called method. See FIG. 47, FIG. 45, and FIG. 46.

Detailed Description Text (26):
3.4 Interface Invocation of Methods

Detailed Description Text (27):
When a method is invoked through an interface declaration, the called method's name and signature is stored as part of the calling method's code representation. The compiler ensures that the object to be operated on has a method of the specified name and signature. However, it is not possible to determine prior to run time the index position within the method table that holds the target method. Thus it is necessary to examine the target object's mtable field, which points to the corresponding MethodTable structure. We follow the MethodTable's methods pointer to find an array of pointers to Method structures. And we search this array for a method that matches the desired name and signature. Once found, we invoke this method. We examine the Method object that represents the target procedure and consult its access.sub.-- flags field to determine if the method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be represented as byte code. See FIG. 50, FIG. 51, FIG. 45, and FIG. 46.

Detailed Description Text (29):
Care must be taken when switching between execution modes. Since mode changes do not occur within methods, all mode changes are associated with calling or returning from a PERC method.

Detailed Description Text (30):
Note that native methods and pvm(), which interprets byte-code methods, use the same stack organization. Thus, calling another method from a native method is the

same as calling the method from within the pvm() interpreter. In both cases, the caller invokes the callee by passing appropriate parameters to one of several available invocation routines, all of which are written primarily in C. These invocation routines consult internal fields within the Method structure that describes the callee to determine whether the callee is implemented as byte codes, the JIT translation of byte codes, or a native method (See FIG. 46). The invocation routine adjusts the stack and other state information as necessary in order to transfer control to the called method. When the called method returns, the invocation routine restores the stack and other state information to once again support the execution mode of the calling method. To call a byte-code method, the invocation routine saves the offset of the old frame and stack pointers in local C variables, sets up the callee's activation frames (See FIG. 5), and calls pvm(), passing a pointer to the called method's Method structure as the only argument. To call a native method, the invocation routine saves the offsets of the old stack and frame pointers, sets up the native method's activation frames (See FIG. 5), and calls (*Method.native)(). To call a JIT-translated method, the invocation routine sets up the callee's activation frames (See FIG. 5), pushes the current C frame pointer onto the C stack and then saves the current value of the C stack pointer in the c.sub.-- sp field of the currently executing thread's Thread data structure, copies the current values of the .sub.-- psp and .sub.-- npsp variables into machine registers dedicated to these purposes (effectively making the PERC stacks become the run-time execution stacks), and branches to (*Method.jit.sub.-- interface)(), leaving the return address in the stack slot above the top-of-stack entry on the non-pointer stack. See FIG. 94 for an illustration of the non-pointer stack activation frame as it is organized during execution of JIT code.

Detailed Description Text (31):
If the caller is a JIT-translated method, the callee is invoked in all cases by simply branching to the equivalent of (*Method.jit.sub.-- interface)(). A small procedure stub is generated to represent each byte-code and native method in the system. Stub procedures, described below, perform all of the mode switching work that is required in switching execution modes. Note that the JIT-code translation of a static or special (non-virtual) invocation in-lines the address of the callee's code so that the corresponding Method structure does not need to be consulted at run time.

Detailed Description Text (33):
To invoke another method from within pvm(), we call one of invokeStatic(), invokeSpecial(), invokeVirtual(), or invokeInterface(), as described in Table 1.

Detailed Description Text (34):
4.1.1 Invocation of Virtual Methods

Detailed Description Text (35):
From within the implementation of the pvmo and within native methods, the standard protocol for invoking other methods depends on the type of the call. A virtual method invocation vectors to the corresponding code by way of the target object's method table. The object to which the method corresponds is passed implicitly on the run-time stack. To invoke a virtual method, first push a pointer to the target object onto the pointer stack and then push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeVirtual (), passing as arguments pointers to the String objects that represent the class name and the target method's name and signature (See FIG. 49):

Detailed Description Text (36):
void invokeVirtual(String *class.sub.-- name, String *method.sub.-- name.sub.-- and.sub.-- sig); Note that invokeVirtual() must do a string search within the class representation to find the selected method. This is potentially a costly operation and we would prefer to avoid this cost when possible. When byte code is first loaded into our system, we perform this lookup and save the result, represented by

a pointer to a Method structure, within the constant pool. Implementers of native methods may design similar optimizations. There are two mechanisms available to implementers of native methods for the purposes of looking up Method objects: findMethod() and getMethodPtr(). Both of these functions return a pointer to the corresponding Method object. With findMethod(), the desired method is described by a pointer to the known Class object and a String pointer to the method's name and signature. With getMethodPtr(), the desired method is described by String representations of the class name and of the method's name and signature. Prototypes for both functions are provided below:

Detailed Description Text (37):
Method *findMethod(Class *class.sub.-- name, String *method.sub.-- name.sub.-- and.sub.-- sig);

Detailed Description Text (38):
Method *getMethodPtr(String *class.sub.-- name, String *method.sub.-- name.sub.-- and.sub.-- sig);

Detailed Description Text (39):
Both functions return null if the method was not found.

Detailed Description Text (40):
Within the Method structure, information is available which characterizes the number of pointer arguments of this particular method and the offset of this method within the object's method table (see FIG. 22). To invoke a virtual function without incurring the overhead of a string method lookup, use the FastInvokeVirtualo macro, prototyped below (See FIG. 45):

Detailed Description Text (42):
4.1.2 Invocation of Special Methods

Detailed Description Text (43):
Non-virtual method calls resemble virtual method invocations except that the code to be implemented is determined by the declaration (at compile time) rather than by the current instantiation (at run time). There is no need to consult a method table when implementing non-virtual method calls. To invoke a nonvirtual method, call invokeSpecial(), passing as arguments two String objects representing the name of the class and the name and signature of the method within the class, as prototyped below (See FIG. 48):

Detailed Description Text (45):
To optimize the performance of nonvirtual method invocations, first lookup the Method object and remember its location. Then invoke the nonvirtual method by executing the FastInvokeSpecial() macro, prototyped below (See FIG. 45):

Detailed Description Text (46):
void FastInvokeSpecial(Method *);

Detailed Description Text (48):
At the API level, invoking an interface is similar to invoking a virtual or non-virtual method. First push a pointer to the target object onto the pointer stack and then push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeinterface(), passing as arguments String objects representing the name of the class and the name and signature of the method within the class, as prototyped below (See FIG. 50):

Detailed Description Text (50):
To improve the efficiency with which interface methods can be invoked, it is useful to make an educated guess as to where the matching interface might be found within the target object's method table. In most cases, the best guess is the method table

slot at which the match was found the previous time the interface method was invoked. By combining the results of a previous findMethod() invocation with recent execution history, programmers can call interface methods using the FastInvokeInterface() macro, prototyped below (See FIG. 45):

Detailed Description Text (51):
void FastInvokeInterface(int num.sub.-- ptr.sub.-- args, int offset.sub.-- guess, Method *template);

Detailed Description Text (52):
Note in the above that the purpose of the template argument is to allow FastInvokeInterface to determine the name and signature of the method that it must search for in the object found num.sub.-- ptr.sub.-- args slots from the current top-of-stack pointer on the PERC pointer stack.

Detailed Description Text (53):
4.1.4 Invocation of Static Methods

Detailed Description Text (54):
A static method is one that makes use only of information that is associated with the corresponding class (rather than instances of the class). When a static method is invoked, there is no "target object" pushed onto the stack. To call a static method, push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeStatic(), passing as arguments String objects representing the name of the class and the name and signature of the method within the class, as prototyped below (See FIG. 47):

Detailed Description Text (56):
To improve the efficiency with which static methods can be invoked, lookup the corresponding Method object beforehand and remember its location. Then invoke the static method using the FastInvokeStatic() macro, prototyped below (See FIG. 45):

Detailed Description Text (57):
FastInvokeStatic(Method *);

Detailed Description Text (59):
When JIT-translated code invokes a method that is implemented by Java byte code, it is necesary to switch the execution protocol prior to invoking pvm(). Rather than requiring JIT-generated code to check whether this protocol switch is necessary prior to each method invocation, we provide each byte-code method with a stub procedure that honors the JIT execution protocol. This stub procedure switches from JIT to C protocols and then invokes pvm() with appropriate arguments. In more detail, the stub procedure performs the following:

Detailed Description Text (63):
4. Calculates and assigns values to .sub.-- pfp (pointer stack frame pointer) and .sub.-- npfp (non-pointer stack frame pointer), based on the current values of the corresponding stack pointers and the number of arguments of each type. The stack activation frames are arranged as illustrated in FIG. 5. Additionally, we adjust the pointer and non-pointer stack pointers to make room for the local variables that are required to execute the method, as represented by the max.sub.-- ptr.sub.-- locals and max.sub.-- non.sub.-- ptr locals fields of the corresponding Method structure.

Detailed Description Text (64):
5. If the method to be invoked is synchronized, we enter the monitor now, waiting for other threads to exit first if necessary.

Detailed Description Text (66):
7. Calls pvm(), passing as a C argument a pointer to the Method object that

describes the segment of code to be executed.

Detailed Description Text (68):
9. If the invoked method was synchronized, release the monitor now. Note that the pvm() itself takes responsibility for exiting the monitor if the code is aborted by throwing of an exception.

Detailed Description Text (69):
10. The stub procedure then removes all local variables from both PERC stacks, leaving a single pushed quantity on one of the stacks to represent the method's return value. Then it restores the psp, pfp, npsp, and npfp registers if appropriate. (This is our implementation of ReclaimFrames()).

Detailed Description Text (73):
4.3 Native-Method Stubs.

Detailed Description Text (74):
The stub for a native method is identical to the stub for a byte-code method except that the native method is invoked directly rather than invoking pvm() and the stub does not allocate any space on the PERC stacks for the called method's local variables (The native method reserves its own local variable space as needed). Since the target native method's address is known at the time the stub is generated, the native method is invoked directly, without requiring interaction with lookupMethod().

Detailed Description Text (85):
Restrictions. In order to coordinate application processing with garbage collection, it is necessary for authors of native methods and other C libraries to avoid certain "legal" C practices:

Detailed Description Text (143):
Native methods and other C functions run fastest if they avoid frequent copying of values between local variables (stored on the PERC pointer stacks) and C-declared fast pointers. But pointers stored in C-declared variables are not necessarily preserved across preemption of the thread. Thus, it is necessary for the application code to copy from C-declared variables to macro-declared variables before each preemption. The following macros are used to manipulate the pointer stack.

Detailed Description Text (219):
2. When the pvm() (PERC Virtual Machine byte code interpreter) is executing byte-code methods, the method's byte code is represented by a string of bytes. The byte-code instructions are stored in heap memory, suggesting that every instruction fetch needs to incur the overhead of a heap-access macro. To improve the performance of instruction fetching, we allow instruction fetching to bypass the standard heap access macro. Doing so depends on the following:

Detailed Description Text (220):
a. The instruction pointer is represented by a fast pointer declared within the implementation of pvm(). Upon entry into pvm(), this variable is initialized using the GetPC() macro, which expects as arguments a pointer to the ByteString object that represents the method's code and the instruction offset within this method's code (See FIG. 40).

Detailed Description Text (221):
b. Whenever pvm() is to be preempted, or whenever it calls another function that might be preempted, pvm() computes the current instruction offset relative to the beginning of the ByteString object that represents the currently executing method's byte code. We use the GetPCOffset() macro (See FIG. 40).

Detailed Description Text (223):
3. During interpretation of byte-code methods, the constant pool is frequently accessed. Rather than incurring the overhead of a standard heap access macro, we obtain a trustworthy C pointer to the constant pool data structure and refer directly to its contents. For this purpose, we use the GetCP() macro (See FIG. 40). C subscripting expressions based on the value returned by GetCP() are considered valid up to the time at which the thread is next preempted. Following each preemption, the pointer must be recomputed through another application of the GetCP () macro.

Detailed Description Text (225):
Exception handling is implemented using C-style longjmp invocations. Our implementation of try and synchronized statements sets a longjmp buffer for each try and synchronized context that is entered. This imposes an overhead on entry into such contexts even when the corresponding exception is not thrown. An alternative implementation would simply build at compile, byte-code loading, or JIT translation time whatever tables are necessary to allow the exception thrower to search and find the appropriate exception handler whenever exceptions must be thrown. We currently prefer our technique for real-time execution because it allows both entry into try and synchronized statements and the throwing of exceptions to be time deterministic. As a future optimization, we intend to minimize the amount of information that is saved by each setjmp() invocation. We also intend to further study these issues and may ultimately decide to switch to the alternative table-driven implementation of the exception throwing mechanism.

Detailed Description Text (226):
With native and JIT-generated code, entry into a try clause results in the creation of a new jmpbuf to serve as the default exception handler. For methods implemented in byte code, a jmpbuf is allocated upon entry into method's code. The default exception handler is identified by a thread state variable. If we leave the try clause through normal (unexceptional) termination (or leave the body of a byte-code method that includes a try clause), we restore the default exception handler to whatever value it held before we entered into the body of this try statement (or byte-code method). To implement this, we link the PERC-specific jump buffers within the current C stack (as a local variable).

Detailed Description Text (227):
In the case of native methods, the "body" of the try statement must be implemented as a function call. When an exception is raised, the thrown object is stored in a state variable associated with the currently active thread (current.sub.--exception) and the LongJmp() macro is invoked. When the try statement catches the LongJmp() invocation, it checks first to see if this exception handler desires to handle the thrown exception. If so, we handle it here. If not, we simply throw the exception to the surrounding exception handler. See FIG. 27.

Detailed Description Text (228):
Native methods. Upon entry into a block of code that represents either a try block or a synchronized block, we save the previous value of the exception handling jump buffer in a local variable and set a stack-allocated jump buffer to represent this block's exception handler. Whenever an exception is raised, it performs a longjmp to the currently active exception handler. When this exception handler catches the longjmp invocation, it handles it if possible. Otherwise, it simply forwards the exception to the outer nested exception handling context.

Detailed Description Text (230):
The PERC virtual machine. The PERC interpreter (virtual machine) is invoked once for each method to be interpreted. If the method to be interpreted contains synchronized or try blocks, a jump buffer is initialized according to the same protocol described above.

Detailed Description Text (233):
1. The virtual machine implementation: pvm() deserves special treatment since its performance is so critical. The caller of pvm(), which may be a stub procedure for a particular byte-code method, must set up the PERC stack activation frames for execution of this pvm(). Upon return from pvm( ), the caller removes the activation frames from the stacks. In place of the local arguments, the caller leaves a single placeholder to represent the return value on whichever PERC stack is appropriate, or leaves no placeholder if the method is declared as returning void. The activation frame maintenance performed by a stub procedure is described in Section 4.2 on page 18. The activation frame maintenance performed by an invokeVirtual(), invokeSpecial(), invokestatico, or invokeInterface() function is described later in this section under subheadings "PrepareJavaFrames()" and "ReclaimFrames()".

Detailed Description Text (234):
If the implementation of pvm() desires to make use of local variables (required by the C programmer) in addition to the local variables declared as local variables within the PERC method, it should use the AllocLocalPointers() macro (See subheading "AllocLocalPointers()" later in this section).

Detailed Description Text (235):
2. Native methods: Like the virtual machine, each invocation of a native method must be preceded by the preparation of PERC stack activation frames. The format of the activation frames and the protocol for allocation of local pointers is exactly the same for native methods as for pvm().

Detailed Description Text (236):
3. Fast procedures: A fast procedure is a C function called by the pvm(), native methods, or other fast or slow procedures, that is by design, not preemptible. Arguments to a fast procedure are passed on the C stack using traditional C argument passing conventions. Fast procedures should not attempt to access information placed on the PERC stacks by the calling context. (The current PERC implementation makes a non-portable exception to this rule in the implementation of the FastInvokeo macros described in Section 2.0.) This is because the code generated by a custom C compiler that is designed to support accurate garbage collection of C code may place information onto the PERC stacks that would obscure the data placed there by outer contexts.

Detailed Description Text (237):
4. Slow procedures: A slow procedure is a preemptible C function called by the pvm (), native methods, or other slow procedures. In order to make the C function preemptible, it is necessary to coordinate with certain protocols:

Detailed Description Text (242):
AllocLocalPointers(). The AllocLocalPointers() macro may be used only within the implementations of the pvm() and of native methods. If present, the AllocLocalPointers() macro must follow the last local declaration and precede the first line of executable code. The parameterization is as follows:

Detailed Description Text (259):
The values passed as parameters to the PrepareJavaFrames() macro are determined by examining the corresponding fields of the Method structure that is to be invoked.

Detailed Description Text (262):
PrepareNativeFrames(). In preparation for calling a native method, as is done within the invoke routines (invokeVirtual(), invokeSpecial(), invokeStatic(), and invokeInterface()) and within byte-code stubs, it is necessary to construct the activation frames for the PERC pointer and non-pointer stacks. This is done by executing the PrepareNativeFrames() macro, with parameters similar to what was described above for BuildFrames():

Detailed Description Text (264):
The values passed as parameters to the PrepareNativeFrames() macro are determined by examining the corresponding fields of the Method structure that is to be invoked. Note that, unlike byte-code methods, the Method structure has no representation of the number of local variables or the amount of stack growth that will need to be supported during execution of the native method. Once inside the native method, local variables and additional stack growth is specified through the use of the AllocLocalPointers() macro described above.

Detailed Description Text (267):
ReclaimFrames(). Upon return from a native method or pvm(), the activation frames constructed by PrepareJavaFrames() or PrepareNativeFrames() must be dismantled. This is implemented by the ReclaimFrames() macro, parameterized the same as DestroyFrames():

Detailed Description Text (273):
The PERC Virtual Machine describes the C function that interprets Java byte codes. This C function, illustrated in FIG. 68, is named pvm(). The single argument to pvm () is a pointer to a Method structure, which includes a pointer to the byte-code that represents the method's functionality. Each invocation of pvm() executes only a single method. To call another byte-code method, pvm() recursively calls itself. Note that pvm() is reentrant. When multiple Java threads are executing, each thread executes byte-code methods by invoking pvm() on the thread's run-time stack.

Detailed Description Text (274):
The implementation of pvm() allocates space on the PERC pointer stack for three pointer variables. These pointers, known by the symbolic names pMETHOD, pBYTECODF, and pCONSTANTS, represent pointers to the method's Method structure, the StringOfBytes object representing its byte code, and the constant-pool object representing the method's constant table respectively. During normal execution of pvm(), the values of these variables are stored in the C locals method, bytecode, and cp respectively. Before preemption, and before calling preemptible functions, pvm() copies the contents of these C variables onto the PERC pointer stack. In preparation for executing the byte codes representing a byte-code method, pvm() checks to determine if the method has any exception handlers. If the method is synchronized, the lock will have been obtained by the fastInvokeo routine prior to calling pvm() (see FIG. 46). However, fastInvoke() does not set an exception handler to release the lock if the code is aborted by the raising of an exception. For this reason, pvm() sets an exception handler if the method is synchronized, so that it can release the lock before rethrowing the exception to the surrounding context.

Detailed Description Text (277):
1. The current.sub.-- method local variable is non-zero if and only if this pvm() invocation is currently executing. Each time pvm() calls another function, current.sub.-- method is set to 0. When the function returns, current.sub.-- method is set to 1. The purpose of this variable is to determine whether a caught exception was thrown by this pvm() invocation or by some other function which had been called by pvm(). If the exception was thrown by pvm(), next.sub.-- pc.sub.-- offset is not valid and must be computed from the current value of next.sub.-- c. See FIG. 68.

Detailed Description Text (278):
2. The next pc.sub.-- offset variable, which represents the byte offset within the current method's byte code of the next instruction to be executed within this method, is passed to findExceptionHandler().

Detailed Description Text (279):
3. findExceptionliandler() searches within the current method's exception table for the exception handler that corresponds to the current execution point within the

method's byte code.

Detailed Description Text (281):
5. If no exception handler is found, pvm() first releases the monitor lock if this method was synchronized and then it rethrows the exception to the surrounding exception handling context.

Detailed Description Text (287):
The JSR instruction (See FIG. 73) jumps to a subroutine by branching to the byte-code instruction obtained by adding the two-byte signed quantity which is part of the instruction encoding to the current value of the program counter and pushing the return address onto the non-pointer stack. Note that the return address is represented as the integer offset within the current method's byte code rather than an actual pointer. This is because the garbage collector does not deal well with pointers that refer to internal addresses within objects rather than to the objects' starting addresses. Note also that the JSR instruction also invokes the PVMPremptionPoint() macro.

Detailed Description Text (288):
The RET instruction (See FIG. 74) returns from a subroutine by fetching the return address from the local integer variable found at the offset specified by the byte-code instruction's one-byte immediate operand. The return address is represented as an offset within the byte-code method, so it is converted into an actual instruction address by using the GetPC() macro. Note also that the RET instruction invokes the PVMPreemptionPoint() macro.

Detailed Description Text (291):
The IRETURN instruction (See FIG. 77) is used to return an integer from the currently executing method. This instruction pops the integer value to be returned from the top of the non-pointer stack and stores the integer value into the 0th slot of the non-pointer stack's current activation frame. Then it breaks out of the interpreter loop by using a goto statement.

Detailed Description Text (292):
The GETSTATIC.sub.-- QNP8 instruction (See FIG. 78) gets an 8-bit non-pointer value from the static area of the class corresponding to the field that is stored in the constant-pool table at the offset specified by this instruction's one-byte immediate-mode operand. The value fetched from the static field is pushed onto the non-pointer stack.

Detailed Description Text (293):
The PUTFIELLD.sub.-- Q instruction stores a value (provided on one of the PERC stacks) into the specified field of a particular object. A pointer to the object that contains the field is passed on the pointer stack. The two-byte immediate operand of this instruction indexes into the constant pool to find a 4-byte integer value. This integer value encodes the offset of the field within the object as the least significant 29 bits, an encoding of the number of bits to be updated if the field is not a pointer in the next two most significant bits, and a flag distinguishing pointer fields in the most significant bit. The implementation of this instruction is illustrated in FIG. 79.

Detailed Description Text (294):
The INVOKEVIRTUAL.sub.-- FQ instruction (See FIG. 80) invokes a virtual function. The method-table index is encoded as the first immediate-mode byte operand and the number of pointer arguments is encoded as the second immediate-mode byte operand. Note that most of the work associated with invoking the virtual method is performed by the FastInvokeVirtual() macro, which is illustrated in FIG. 45. Note also that pvm() saves and restores its state surrounding the method invocation.

Detailed Description Text (295):

The implementation of INVOKESPECIAL.sub.-- Q (See FIG. 81) closely resembles INVOKEVIRTUAL.sub.-- Q. The method to be invoked is obtained by fetching the constant-pool entry found at the index position identified by the two-byte immediate operand of this instruction. INVOKESTATIC.sub.-- Q (See FIG. 82) is encoded the same as INVOKESPECIAL.sub.-- Q. The implementation is very similar.

Detailed Description Text (296):
Invocation of interfaces is performed by the INVOKEINTERFACE.sub.-- Q instruction (See FIG. 83). Invoking interfaces is inherently more complicated than the other forms of invocation because the method table of the target object must be searched for a method with a matching name and signature. It is not generally possible to map the name and signature to an integer index prior to execution of the instruction. The immediate-mode operands to this instruction are (1) a one-byte index into the constant pool table to obtain a pointer to a method that has the desired name and signature, (2) a one-byte operand representing the number of pointer arguments passed to the interface method, and (3) a one-byte guess as to the offset within the target object's method table at which the target method will be found. See the definition of the FastInvokeInterface() macro in FIG. 45.

Detailed Description Text (297):
The NEW.sub.-- Q instruction (See FIG. 84) allocates a new object. This instruction takes a two-byte immediate-mode operand, which is an index into the constant pool. The corresponding entry within the constant pool is a pointer to the Class object (See FIG. 16) that describes the type of the object to be allocated. The newly allocated object is pushed onto the pointer stack.

Detailed Description Text (299):
The ANEWARRAY.sub.-- Q instruction (See FIG. 86) allocates a new array of pointers. The type of the array entry is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the corresponding Class object. The size of the array is passed as an integer on the non-pointer stack. The newly allocated array is pushed onto the pointer stack.

Detailed Description Text (301):
The CHECKCAST.sub.-- Q instruction (See FIG. 88) ensures that the top pointer stack element is of the appropriate type, where appropriate type is defined to mean that the type of the stack element is derived from the "desired" type. If it is not, this instruction throws an exception. The desired type is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the Class object that represents the desired type. Note that the NULL value is considered to match any reference type. If the top pointer stack value is of the appropriate type, the value is left on top of the pointer stack.

Detailed Description Text (302):
The INSTANCEOF.sub.-- Q instruction (See FIG. 89) removes the top pointer stack element and checks to see if it is of the appropriate type, where appropriate type is defined to mean that the type of the stack element is the "desired" type. If it is, this instruction pushes a 1 onto the non-pointer stack. If it isn't, this instruction pushes a 0 onto the non-pointer stack. The NULL value is considered to be of the appropriate type. The desired type is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the Class object that represents the desired type.

Detailed Description Text (319):
Obtaining a hash value. When application code desires to obtain the hash value of a particular object, it invokes the native hashCode() method. This method consults the object's lock field. If this field is NULL, this method allocates a HashLock object, initializes its hash.sub.-- value field to the next available hash value, and initializes the object's lock pointer to refer to the newly alocated HashLock object. Then it returns the contents of the hash.sub.-- value field. If the lock

field is non-NULL, hashCode() consults the hash.sub.-- value field of the corresponding HashLock object to determine whether a hash value has already been assigned. If this field has value 0, hashCode() overwrites the field with the next available hash value. Otherwise, the hash value has already been assigned. In all cases, the last step of hashCode() is to return the value of the hash.sub.-- value field.

Detailed Description Text (351):
The Java implementation of the TaskDispatcher class is illustrated in FIG. 53. This class is represented by a combination of Java and native methods. The native methods provide an interface to services provided by the underlying operating system. Note that TaskDispatcher extends Thread.

Detailed Description Text (353):
When the TaskDispatcher object is instantiated, the constructor invokes the initDispatcher() native method, illustrated in FIG. 54. This invention disclosure describes the implementation for the Microsoft Windows WIN32 API. The initDispatcher() method performs the following:

Detailed Description Text (362):
When the TaskDispatcher's run method is invoked (automatically by the PERC run-time system since TaskDispatcher extends Thread), we perform the following:

Detailed Description Text (363):
1. Invoke the startDispatcher() native method (See FIG. 54). This native method resumes the watchdog thread, allowing it to begin running. This is necessary because when the watchdog thread was originally created, it was configured to be in a suspended state.

Detailed Description Text (368):
c. Executing the selected thread for a 25 ms time slice by calling the runThreado native method (See FIG. 54).

Detailed Description Text (369):
Note that implementation of task priorities is provided by the nrt.sub.-- ready.sub.-- q object. Its getNextThread() method always returns the highest priority thread that is ready to run. Note also that it would be straightforward to modify this code so that the duration of each thread's time slice is variable. Some thread's might require CPU time slices that are longer than 25 ms and others might tolerate time slices that are shorter. runThread() (See FIG. 54) performs the following:

Detailed Description Text (376):
a. An event is triggered by the watchdog task, or by the task executing its relinquish() method (See FIG. 54). This event will be triggered if the dispatched task blocks (on I/O or sleep, for example).

Detailed Description Text (386):
12. Restores the state of the dispatcher task (and returns to the Java method that invoked the native runThread() method.

Detailed Description Text (435):
1. The static region represents memory that is not relocated by the garbage collector. In general, this region comprises C stack segments for use by threads, segments of code produced by the JIT compiler, and stubs for byte-code and native methods.

Detailed Description Text (438):
We intend for byte codes to be stored as part of the dynamic heap. This means they will be relocated as necessary on demand. However, the results of JIT compilation

are stored in static memory. Note that each JIT-translated method is represented by a Method object which is stored in the garbage collected heap. The finalize() method for the Method object explicitly reclaims the static memory that had been reserved for representation of the method's JIT translation.

Detailed Description Text (443):
There are two static memory segments supported by our run-time system. Static memory segments are never relocated and are not currently garbage collected. The static data region represents the code produced by the JIT translator, native-method and byte-code-method stubs, and C stacks.

Detailed Description Text (445):
To allocate code segment memory for the JIT translation of byte codes, for byte-code stubs, and for dynamically constructed shared signatures, use allocCS(), prototyped below:

Detailed Description Text (456):
struct GenericConstant.sub.-- Info **constant.sub.-- pool;

Detailed Description Text (495):
Every PERC object begins with two special fields representing the object's lock and method tables respectively. See FIG. 23 for the declaration of MethodTable. The method table's first field is a pointer to the corresponding Class object. The second field is a pointer to an array of pointers to Method objects. The third field is a pointer to the JIT-code implementation of the first method, followed by a pointer to the JIT-code implementation of the second method, and so on. The pointers to JIT-code implementations may actually be pointers only to stub procedures that interface JIT code to byte-code or native-code methods.

Detailed Description Text (496):
Allocation routines. When allocating memory from within a native method, the programmer provides to the allocation routine the address of a signature rather than simply the size of the object to be allocated. The Signature pointer passed to each allocate routine must point to a statically allocated Signature structure. The implementation of the PERC virtual machine allocates a static signature for each class loaded. Once this static signature has been created, all subsequent instantiations of this class share access to this signature.

Detailed Description Text (539):
1. Except for from-space, each demi-space maintains its own free pool. Further, each space remembers the total amount of memory represented by its free pool. See FIG. 95.

Detailed Description Text (540):
2. Each free pool is organized as several linked lists, one for objects of size 4, 8, 16, 32, 64, 128, . . . , 512K words. Free segments of sizes that don't exactly match one of the corresponding list sizes are placed on the list for the next smaller size. Thus, every "object" on the free list for size 64 is between 64 and 127 words large, inclusive. Note that the sizes of the objects represented by the different free lists need not be powers of two. For example, the fibonacci sequence may be a better choice.

Detailed Description Text (549):
5. At the time garbage collection begins (flip time), we sort the mark-and-sweep spaces according to amounts of available memory. Our preference is to allocate free memory from the space that is already most full. We link the free lists of the two mark-and-sweep free pools and the to-space free pool to reflect this preference. We always put to-space as the last region on this list, because we prefer to allocate from the mark-and-sweep regions if they have space available to us.

Detailed Description Text (550):
6. To allocate a heap object from a region's free pool, select the first (smallest) free list that is known to contain free segments of sufficiently large size. If the free list is not empty, remove the first segment on that free list, divide that segment into two smaller segments with one being of the requested size and the other being returned to the appropriate free list (if the free segment is sufficiently large), and return the allocated memory. If the selected free list is empty, repeat this algorithm on the next larger size free list (until there are no larger free lists to try).

Detailed Description Text (553):
In Java, programmers can specify an action to be performed when objects of certain types are reclaimed by the garbage collector. These actions are specified by including a non-empty finalize method in the class definition. Such objects are said to be finalizable. When a finalizable object is allocated, the two low order bits of the Activity Pointer are set to indicate that the object is finalizable. The 0.times.01 bit, known symbolically as FINAL.sub.-- LINK, signifies that this object has an extra Finalize Link field appended to the end of it. The 0.times.02 bit, known symbolically as FINAL.sub.-- OBJ, signifies that this object needs to be finalized. After the object has been finalized once, its FINAL.sub.-- OBJ is cleared, but its FINAL.sub.-- LIINK bit remains on throughout the object's lifetime.

Detailed Description Text (555):
The run-time system includes a background finalizer thread which takes responsibility for incrementally executing the finalizers associated with all of the objects reachable from the Finalizees root pointer. Following execution of the finalizer method, the finalizes object is removed from the finalizes list and its Activity Pointer field is overwritten with a reference to the corresponding Activity object. Furthermore, we clear the FINAL.sub.-- OBJ so we don't finalize it again. Optionally, each real-time activity may take responsibility for timely finalization of its own finalizee objects. Typically, this is done within an ongoing real-time thread that is part of the activity's workload.

Detailed Description Text (570):
d. Once to- and from-space have been selected and initialized, the free pools of the remaining regions are linked together in increasing order of amount of free memory. The free pool of the current to-space is linked onto the end of this list. Every request for new memory allocation will be satisfied by searching the free spaces of the various regions in the order determined by these links. FIG. 95 illustrates the results of linking the independent free lists into a single global free pool.

Detailed Description Text (613):
Before starting the second pass, we identify each of the entry points to the method. We consider the first basic block in the method to be the main entry point. Additionally, we consider the starting block for each finally statement to represent an entry point. And further, we consider the starting block for each exception handler to represent an entry point. Exception handlers are identified in the method's code attribute, in the field named exception.sub.-- table. The relevant data structures are described in "The Java Virtual Machine Specification", written by T im Lindholm and Frank Yellin, published in 1996.

Detailed Description Text (615):
1. The offsets within the method's byte code of the instructions that represent the start and end of the basic block.

Detailed Description Text (629):
9.2.1 Constant-Pool Optimizations

Detailed Description Text (630):
Most of the operations that access the constant pool can be replaced with fast variants. When a Java class is loaded into the Java virtual machine, all of the constants associated with each method are loaded into a data structure known as the constant pool. Because Java programs are linked together at run time, many constants are represented symbolically in the byte code. Once the program has been loaded, the symbolic values are replaced in the constant pool with the actual constants they represent. We call this process "resolving constants." Sun Microsystems Inc.'s descriptions of their Java implementation suggest that constants should be resolved on the fly: each constant is resolved the first time it is accessed by user code. Sun Microsystems Inc.'s documents further suggest that once an instruction making reference to a constant value has been executed and the corresponding constant has been resolved, that byte code instruction should be replaced with a quick variant of the same instruction. The main difference between the quick variant and the original instruction is that the quick variant knows that the corresponding constant has already been resolved.

Detailed Description Text (631):
In our system, we resolve the entire constant pool when the class is loaded. Furthermore, we examine all of the byte codes corresponding to each method and replace them as necessary to represent the appropriate quick variants. Our implementation differs (apparently) from Sun Microsystems Inc.'s in that we do not need to dedicate byte codes to represent the slow variants of these instructions. In our system, all constants are known to be resolved prior to execution of the corresponding byte codes.

Detailed Description Text (633):
item found on the specified one-byte indexed position within the constant pool table onto the stack. If this item is an object pointer, we need to push the pointer value onto the pointer stack. If this item is not a pointer, we push its value onto the non-pointer stack. We use code 18 to represent ldcl.sub.-- np, which loads a non-pointer constant onto the non-pointer stack. We use code 255 to represent ldc1.sub.-- p, which loads a pointer constant onto the pointer stack. ldc2. This operation is represented by code 19. This instruction pushes the item found on the specified two-byte indexed position within the constant pool table onto the stack. If this item is an object pointer, we need to push its value onto the pointer stack. If this item is not a pointer, we push its value onto the non-pointer stack. We use code 19 to represent ldc2.sub.-- np, which loads a non-pointer constant onto the non-pointer stack. We use code 254 to represent ldc2.sub.-- p, which loads a pointer constant onto the pointer stack.

Detailed Description Text (634):
Putfield. This operation is represented by code 181. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. The loader replaces this code with one of the following:

Detailed Description Text (635):
1. putfield.sub.-- q encoded as 181: We replace the constant-pool entry with an integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less than 64 Kbytes and can thus be represented in the 16-bit immediate operand representing an unsigned integer quantity.

Detailed Description Text (641):
Getfield. This operation is represented by code 180. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. This code is replaced with one of the following:

Detailed Description Text (642):
1. getfield.sub.-- q encoded as 180: We replace the constant-pool entry with a 32-bit integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less than 64 Kbytes and can thus be represented in the 16-bit immediate operand representing an unsigned integer quantity.

Detailed Description Text (648):
Putstatic. This operation is represented by code 179. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a pointer to the Field structure that describes the field to be updated. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (654):
Getstatic. This operation is represented by code 178. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a pointer to the Field structure that describes the field to be fetched. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (660):
Anewarray. This operation is represented by code 189. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool. When resolved, the selected constant must be a class. The result of this operation is a newly allocated array of pointers to the specified class. The loader replaces this instruction with anewarray.sub.-- q, which is also encoded as operation 189. This instruction differs from anewarray in that it does not need to resolve the constant entry. Rather, it assumes that the specified slot of the constant pool holds a pointer directly to the corresponding class object.

Detailed Description Text (661):
Multianewarray. This operation is represented by code 197. It takes two immediate-mode byte operands to represent a 16-bit constant pool index and a third immediate-mode byte operand to represent the number of dimensions in the array to be allocated. The index position is handled the same as for anewarray. The loader replaces this instruction with multianewarray.sub.-- q, which is encoded as operation 197. This instruction differs from multianewarray in that it does not need to resolve the constant entry. Rather, it assumes that the specified slot of the constant pool holds a pointer directly to the corresponding class object.

Detailed Description Text (662):
Invokevirtual. This operation is represented by code 182. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method signature, including its name. If the method-table index of the corresponding method is greater than 255 or if the number of pointer arguments is greater than 255, the loader replaces this instruction with invokevirtual.sub.-- q, encoded as operation 182. Otherwise, the loader replaces this instruction with invokevirtual.sub.-- fq, encoded as operation 233.

Detailed Description Text (663):
With the invokevirtual.sub.-- fq instruction, the first immediate-mode byte operand represents the method table index and the second immediate-mode byte operand represents the number of pointer arguments.

Detailed Description Text (664):
With the invokevirtual.sub.-- q instruction, the two immediate-mode operands represent the same 16-bit index into the constant pool table as with the original invokevirtual instruction. However, this entry within the constant pool table is overwritten with a pointer to the Method structure that describes this method. (Note that both invokevirtual and invokespecial may share access to this same entry in the constant pool. In fact, there is no difference between the implementations of invokespecial.sub.-- q and invokestatic.sub.-- q in certain frameworks.)

Detailed Description Text (665):
invokespecial. This operation is represented by code 183. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method signature, including its name. This instruction is replaced with invokespecial.sub.-- q, encoded as 183. With the invokespecial.sub.-- q instruction, the selected constant pool cntry is replaced with a pointer to the Method structure that describes this method. (Note that both invokevirtual and invokespecial may share access to this same entry in the constant pool.)

Detailed Description Text (666):
invokestatic. This operation is represented by code 184. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method's class and signature, including its name. The loader replaces this instruction with invokestatic.sub.-- q, encoded as 184. The distinction of invokestatic.sub.-- q is that the selected constant pool entry is a pointer to the Method structure that describes this method.

Detailed Description Text (667):
Invokeinterface. This operation is represented by code 185. The instruction takes a 2-byte constant pool index, a one-byte representation of the number of arguments, and a one-byte reserved quantity as immediate-mode operands. The corresponding constant-pool entry represents the method's signature. The loader replaces this instruction with invokeinterface.sub.-- q, encoded as 185. The distinction of invokeinterface.sub.-- q is that the constant pool entry is overwritten with a pointer to a Method structure that represents the name and signature of the interface method and the reserved operand is overwritten with a guess suggesting the "most likely" slot at which the invoked object's method table is likely to match the invoked interface. If this slot does not match, this instruction searches the object's method table for the first method that does match. On each execution of invokeinterface.sub.-- q, the guess field is overwritten with the slot that matched on the previous execution of this instruction.

Detailed Description Text (669):
New. This operation is represented by code 187. The instruction takes a 2-byte

constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with new.sub.-- q, also encoded as 187. The distinction of new.sub.-- q is that the constant pool entry is replaced with a pointer to the resolved class object.

Detailed Description Text (670):
Checkcast. This operation is represented by code 192. The instruction takes a 2-byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with checkcast.sub.-- q, also encoded as 192. The distinction of checkcast.sub.-- q is that the constant pool entry is replaced with a pointer to the resolved class object.

Detailed Description Text (671):
Instanceof. This operation is represented by code 193. The instruction takes a 2-byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with instanceof.sub.-- q, also encoded as 193. The distinction of instanceof.sub.-- q is that the constant pool entry is known to have been replaced with a pointer to the resolved class object.

Detailed Description Text (681):
dup .times.1. This operation is represented by code 90. It duplicates the top stack item, shifts the top two stack items up one position on the stack, and inserts the duplicated top stack item into the newly vacated stack position. Note that the translation of this instruction depends on the types of the top two stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 01 represents the condition in which the top stack element is a pointer and the next entry is a non-pointer. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (686):
dup .times.2. This operation, encoded as 91, duplicates the top stack entry, shifts the top three stack entries up one stack position, and inserts the duplicated stack entry into the newly vacated stack position. Note that the translation of this instruction depends on the types of the top three stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 001 represents the condition in which the top stack element is a pointer and the next two entries are non-pointers. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (695):
dup2 .times.1. This operation, encoded as 93, duplicates the top two stack entries, shifts the top three stack entries up two stack positions, and inserts the duplicated stack entries into the newly vacated stack slots. Note that the translation of this instruction depends on the types of the top three stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 001 represents the condition in which the top stack element is a pointer and the next two entries are non-pointers. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (704):
dup2 .times.2. This operation is represented by code 94. It duplicates the top two stack items, shifts the top four stack items up two positions on the stack, and inserts the duplicated stack items into the newly vacated stack positions. Note that the translation of this instruction depends on the types of the top four stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 0001 represents the condition in which the top stack element is a pointer and the next three are non-pointers. Each combination of four binary digit type codes represents a decimal number. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (727):
JIT-generated methods usc only the PERC pointer and non-pointcr stacks. All pointer information is stored on the pointer stack and all non-pointer information is stored on the non-pointer stack. The non-pointer activation frame is illustrated in FIG. 94. The pointer activation frame is identical except that there is no return address stored in the pointer activation frame.

Detailed Description Text (728):
Within a JIT-generated method, all local variables, including incoming and outgoing arguments are referenced at fixed offsets from the register that represents the corresponding stack pointer. There is no need for a frame pointer because the stack pointer remains constant throughout execution of the method.

Detailed Description Text (729):
Note that the JIT method's prologue subtracts a constant value from the stack pointer and the JIT method's epilogue adds the same constant value to the stack pointer.

Detailed Description Text (730):
When JIT-compiled methods invoke byte-code or native-codc methods, the corresponding byte-code stub sets up the frame and stack pointers necessary for execution of the corresponding C routines. Additionally, the return address is removed from the non-pointer stack and stored temporarily in a C local variable within the stub procedure.

Detailed Description Text (731):
Within a JIT-compiled method, machine registers are partitioned so that certain registers are known to only contain base pointers and other machine registers are known to only contain non-pointers. An additional class of registers may contain derived pointers which refer to the internal fields of particular objects. Each derived-pointer register is always paired with a base-pointer register which is known to identify the starting address of the corresponding object. Otherwise, the derived-pointer register holds the NULL value.

Detailed Description Text (732):
During execution of JIT-compiled methods, the thread status is set to JIT.sub.--EXECUTION. This signals to the task dispatcher that the task can be preempted at any time.

Detailed Description Text (733):
The JIT compiler provides special translations of exception handling contexts, so that the equivalents of setjmp() and longjmp() are specialized for the local execution environment. Rather than saving and restoring all machine registers, only those registers that are actually live on entry into the exception handling context are saved and restored.

Detailed Description Text (735):
The ROMizer tool analyzes and verifies byte code and performs byte-code and constant-pool transformations described in the previous section. Additionally, the ROMizer tool supports standard compiler transformations designed to optimize the performance of executed code. These optimizations include in-lining of small and/or performance critical methods, relocation of loop-invariant code outside the loop, and constant folding (including elimination of redundant array subscript checking).

Detailed Description Text (737):
The ROMizer also has the ability to translate bytc code to native code, by applying the JIT compiler to the byte code prior to constructing the ROM load image. In performing these translations, additional optimizations are applied. These include global register allocation, optimal instruction selection, and pipeline scheduling.

Detailed Description Text (742):
3. All byte codes are pre-transformed into the PERC extended instruction set and all references to the constant pool have already been resolved.

Detailed Description Text (746):
5. The machine code that results from JIT compilation is stored within a PERC object whose signature is array of integer. All of the code for JIT-translated methods is stored in a single array of integer object.

Detailed Description Text (749):
1. The code used in the implementation of the ROMizer tool to read in a Java class file, verify the validity of the byte code, and transform the byte code into the PERC instruction set is the exact same code that is used by the PERC implementation to support dynamic (on-the-fly) loading of new byte-code functionality into the PERC virtual machine.

Detailed Description Paragraph Table (1):
TABLE 1 _____ Mode Changes Between Different Method Implementations Called Function Calling Function Byte Code JIT Code Native Method _____ Byte Code invokeStatic() invokeStatic () invokeStatic() (pvm()) invokeSpecial() invokeSpecial() invokeSpecial() invokeVirtual() invokeVirtual() invokeVirtual() invokeInterface invokeInterface invokeInterface () () () JIT Code Byte code stub Direct call Native method stub Native Method invokeStatic() invokeStatic() invokeStatic() invokeSpecial() invokeSpecial()

CLAIMS:

1. A real-time virtual machine method (RTVMM) for implementing real-time systems and activities, the RTVMM comprising the steps:

implementing an O-OPL program that can run on computer systems of different designs, an O-OPL program being based on an object-oriented programming language (O-OPL) comprising object type declarations called classes, each class definition describing the variables that are associated with each object of the corresponding class and all of the operations called methods that can be applied to instantiated objects of the specified type, a "method" being a term of art describing the unit of procedural abstraction in an object-oriented programming system, an O-OPL program comprising one or more threads wherein the run-time stack for each thread is organized so as to allow accurate identification of type-tagged pointers contained on the stack without requiring type tag information to be updated each time the stack's content changes, the O-OPL being an extension of a high-level language (HLL) exemplified by Java, HLL being an extension of a low-level language (LLL) exemplified by C and C++, a thread being a term of art for an independently-

executing task, an O-OPL program being represented at run time by either O-OPL byte codes or by native machine codes.

3. The RTVMM of claim 1 wherein an O-OPL program comprises one or more classes represented in read-only memory, the methods thereof h aving been converted into O-OPL byte codes prior to run time.

4. The RTVMM of claim 1 wherein an O-OPL program comprises one or more classes represented in read-only memory, the methods thereof having been converted into native machine language prior to run time.

5. The RTVMM of claim 1 wherein a byte-code O-OPL method is an O-OPL method represented at run time by O-OPL byte codes, a byte-code O-OPL method being written in O-OPL, an O-OPL, method represented at run time by native machine codes being either a native O-OPL method or a native-translated O-OPL method, a native O-OPL method being written in LLL, a native-translated O-OPL method being written in HLL, the implementing step comprising the steps:

compiling the byte-code O-OPL methods into HLL byte codes and transforming the HLL byte codes into O-OPL byte codes;

compiling the native O-OPI, methods into native machine codes;

compiling the native-translated O-OPL methods into HLL byte codes and compiling HLL byte codes into native machine codes.

6. The RTVMM of claim 1 wherein a calling function is a native-translated O-O-OPL method and the called function is a byte-code method, a native-translated O-OPL method being an O-OPL method written using byte codes which are translated into native machine language at the time of execution, a byte-code method being a method written using O-OPL or HLL and translated into O-OPL, byte codes prior to execution, the implementing step comprising the steps:

providing each byte-code method with a stub procedure which honors the native-translated method execution protocol, the stub procedure switching from native-translated method to O-OPL byte code interpretation protocols and then invoking an O-OPL, interpreter.

7. The RTVMM of claim 6 wherein the stub procedure switches back to the native-translated O-OPL mode when the O-OPL, interpreter returns.

8. The RTVMM of claim 1 wherein a calling function is a native-translated O-OPL method and the called function is a native method, a native-translated O-OPL method being a method written using byte codes which are translated into native machine language at the time of execution, a native method being a method written in LLL, the implementing step comprising the steps:

providing each native method with a stub procedure which honors the native-translated method execution protocol, the stub procedure switching from native-translated method to LLL-code protocols and then invoking the native method.

9. The RTVMM of claim 8 wherein the stub procedure switches back to the native-translated O-OPL mode when the O-OPL interpreter returns.

12. The RTVMM of claim 1 wherein one of the implemented threads is a garbage collection thread that operates asynchronously thereby resulting in the garbage collection thread being interleaved with other threads in arbitrary order, objects subject to garbage collection being either finalizable or non-finalizable, a finalizable object being subject to an action that is performed when the memory space allocated to the finalizable object is reclaimed by the garbage collection

thread, the finalizing action being specified by including a non-empty finalizer method in the class definition, the garbage collection thread being able to distinguish a thread's pointer variables from the thread's non-pointer variables, preemption of a thread being allowed only if the thread is in a state identified as a preemption point, a thread being allowed to hold pointers in variables between preemption points that may not be visible to the garbage collection thread, pointer variables that may not be visible to the garbage collection thread being called fast pointers, pointer variables that are visible to the garbage collection thread being called slow pointers, each LLL, function being identified as either preemptible or non-preemptible.

26. The RTVMM of claim 25 wherein the implementing step comprises the step:

maintaining a free pool of space segments for to-space and for each mark-and-sweep sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "activity pointer" field to specify the size of a free space segment.

28. The RTVMM of claim 27 wherein the implementing step comprises the steps:

maintaining a free pool of space segments for to-space and for each mark-and-sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "signature pointer" field to be used as a backward link to the preceding segment.

37. The RTVMM of claim 24 wherein, after to-space and from-space have been selected at the beginning of a garbage collection cycle, the implementing step comprises the steps:

causing the free pools of memory in the mark-and-sweep spaces and to-space to be linked together into a global free pool, the free pools of the mark-and-sweep spaces being linked in increasing order of amount of free memory, the free pool of to-space being linked to the mark-and-sweep space having the greatest amount of free memory, a request for a new memory allocation being satisfied by the first memory segment of sufficient size found by searching the global free pool according to the linking order.

49. The RTVMM of claim 12 wherein the implementing step comprises the step:

designating portions of memory as a to-space and zero or more mark-and-sweep spaces;

maintaining a free pool of space segments for to-space and for each mark-and-sweep space, a free pool being organized as a plurality of linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping.

50. The RTVMM of claim 49 wherein an object of specified size is to be allocated space in a demi-space by an allocation routine, the allocation routine comprising the steps:

causing the linked list with the smallest size range having space segments equal to or greater than the specified size of the object to be selected from the free pool

of the demi-space;

causing a portion of the space segment equal in size to the object to be allocated to the object;

causing the unallocated portion of the space segment to be returned to the appropriate linked list.

52. The RTVMM of claim 51 wherein the implementing step comprises the steps:

maintaining a free pool of space segments for to-space and for each mark-and-sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "indirect pointer" field to be used as a forward link to the succeeding segment.

54. The RTVMM of claim 53 wherein the implementing step comprises the step:

implementing a finalizer thread that operates in the background and is

responsible for incrementally executing the finalizer methods associated with finalizee objects reachable from the "finalizees" pointer.

55. The RTVMM of claim 54 wherein the finalizer thread comprises the steps:

causing a finalizer method associated with a finalizee object to be executed;

causing the finalizee object to be removed from the associated finalizee list;

causing the "activity pointer" field of the finalizee object to be overwritten with a reference to the allocating object;

causing a "finalize object" bit in the "activity pointer" field of the finalizee object to be cleared indicating that the object has been finalized.

56. The RTVMM of claim 53 wherein the implementing step comprises the step:

implementing a finalizer thread that is part of a real-time activity and is responsible for incrementally executing the finalizer methods associated with finalizee objects associated with the real-time activity and reachable from the "finalizees" pointer.

57. The RTVMM of claim 56 wherein the finalizer thread comprises the steps:

causing a finalizer method associated with a finalizee object to be executed;

causing the finalizee object to be removed from the associated finalizee list;

causing the "activity pointer" field of the finalizee object to be overwritten with a reference to the allocating object;

causing a "finalize object" bit in the "activity pointer" field of the finalizee object to be cleared indicating that the object has been finalized.

74. The RTVMM of claim 1 wherein an O-OPL byte-code loader is used to load a program into a computer system, the implementing step comprises the step:

causing each byte code of an HLL program to be translated into an O-OPL byte code.

76. The RTVMM of claim 1 wherein there is a slow variant and a fast variant of every byte code instruction, a program to be loaded into a computer consisting of one or more slow variants, the implementing step comprising the step:

causing all byte codes corresponding to each method to be examined;

causing the slow variants to be replaced by the quick variants when a class is loaded into a computer.

79. The RTVMM of claim 1 wherein the implementing step comprises the step:

utilizing only O-OPL pointer and non-pointer stacks in executing methods compiled by a JIT compiler, JIT standing for "just in time" and denoting a process for translating HLL, byte codes to native machine language codes on the fly, just in time for its execution, the translation of byte codes to native codes being a form of JIT compiling.

80. The RTVMM of claim 79 wherein a method compiled by a JIT compiler invokes a byte-code or nativc-code method, the implementing step comprises the step:

causing the frame and stack pointers necessary for the execution of the corresponding LLL routines to be set up;

causing the return address to be removed from the non-pointer stack and stored temporarily in an LLL local variable.

81. The RTVMM of claim 79 wherein a method of a thread is being executed, the method having been compiled by a JIT compiler, the implementing step comprising the step:

causing the status of the thread to be set to a value indicating that the thread can be preempted at any time.

82. The RTVMM of claim 79 wherein the implementing step comprises the step:

causing the JIT compiler to provide special translations of exception handling contexts so that only the contents of those registers are saved and restored that are actually live on entry into the exception handling context.

85. The RTVMM of claim 1 wherein the implementing step includes providing a ROMizer tool which produces a load file appropriate for ROM storage, the ROMizer tool comprising the steps:

analyzing and verifying byte code;

performing byte code and constant-pool transformations;

supporting standard compiler transformations designed to optimize the performance of executed code.

88. The RTVMM of claim 85 wherein the implementing step relating to the load file includes the steps:

causing all byte codes to be pre-transformed into an O-OPL instruction set;

causing all references to the constant pool to have been resolved.

First Hit Fwd Refs

End of Result Set

☐ Generate Collection Print

L22: Entry 1 of 1 File: USPT Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A
TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Drawing Description Text (18):
FIG. 17 provides the C declaration of the structure used internal to the PERC implementation to represent a raw class file that has been read into memory. The class-file loader analyzes this object to create an appropriate Class representation.

Detailed Description Text (749):
1. The code used in the implementation of the ROMizer tool to read in a Java class file, verify the validity of the byte code, and transform the byte code into the PERC instruction set is the exact same code that is used by the PERC implementation to support dynamic (on-the-fly) loading of new byte-code functionality into the PERC virtual machine.

First Hit Fwd Refs

End of Result Set

Generate Collection Print

L28: Entry 1 of 1 File: USPT Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A
TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Abstract Text (1):
The invention is a method for use in executing portable virtual machine computer programs under real-time constraints. The invention includes a method for implementing a single abstract virtual machine execution stack with multiple independent stacks in order to improve the efficiency of distinguishing memory pointers from non-pointers. Further, the invention includes a method for rewriting certain of the virtual machine instructions into a new instruction set that more efficiently manipulates the multiple stacks. Additionally, using the multiple-stack technique to identify pointers on the run-time stack, the invention includes a method for performing efficient defragmenting real-time garbage collection using a mostly stationary technique. The invention also includes a method for efficiently mixing a combination of byte-code, native, and JIT-translated methods in the implementation of a particular task, where byte-code methods are represented in the instruction set of the virtual machine, native methods are written in a language like C and represented by native machine code, and JIT-translated methods result from automatic translation of byte-code methods into the native machine code of the host machine. Also included in the invention is a method to implement a real-time task dispatcher that supports arbitrary numbers of real-time task priorities given an underlying real-time operating system that supports at least three task priority levels. Finally, the invention includes a method to analyze and preconfigure virtual memory programs so that they can be stored in ROM memory prior to program.

Brief Summary Text (4):
Java (a trademark of Sun Microsystems, Inc.) is an object-oriented programming language with syntax derived from C and C++. However, Java's designers chose not to pursue full compatibility with C and C++ because they preferred to eliminate from these languages what they considered to be troublesome features. In particular, Java does not support enumerated constants, pointer arithmetic, traditional functions, structures and unions, multiple inheritance, goto statements, operator overloading, and preprocessor directives. In their place, Java requires all constant identifiers, functions (methods), and structures to be encapsulated within

Brief Summary Text (14):
Accurate Garbage Collection, as the term is used in this invention disclosure, describes garbage collection techniques in which the garbage collector has complete knowledge of which memory locations hold pointers and which don't. This knowledge is necessary in order to defragment memory.

Brief Summary Text (16):
Conservative Garbage Collection, as the term is used in this invention disclosure, describes garbage collection techniques in which the garbage collector makes conservative estimates of which memory locations hold pointers. Conservatively, the garbage collector assumes that any memory location holding a valid pointer value (a

legal memory address) contains a pointer. Fully conservative garbage collectors cannot defragment memory. However, partially conservative garbage collectors (in which some pointers are accurately identified) can partially defragment memory.

Brief Summary Text (21):
Fast Pointer is a term specific to this invention disclosure which describes pointers that are implemented using the fastest possible techniques available on a particular computer system. Fast pointers are "normal" pointers as they would be implemented by a typical compiler for the C language.

Brief Summary Text (32):
Pointer is a term of art describing a value held within computer memory or computer registers for the purpose of identifying some other location in memory. The value "points" to a memory cell.

Brief Summary Text (36):
Root Pointer is a term of art describing a pointer residing outside the heap which may point to an object residing within the heap. The garbage collector considers all objects reachable through some chain of pointers originating with a root pointer to be "live."

Brief Summary Text (40):
Slow Pointer is a term specific to this invention disclosure which describes pointers that are implemented in such a way that they provide coordination with a background garbage collection task. Various implementations of slow pointers are possible. In general, fetching, storing, and indirecting through slow pointer variables is slower than performing the same operation on fast pointer variables.

Brief Summary Text (42):
Tending is a term of art describing the garbage collection process of examining a pointer to determine that the object it refers to is live and arranging for the referenced object to be subsequently scanned in order to tend all of the pointers contained therein.

Brief Summary Text (49):
1. Extensions to the standard Java byte code instruction set to enable efficient run-time isolation of pointer variables from non-pointer variables. The extended byte codes are described as the PERC instruction set.

Brief Summary Text (52):
of the PERC instruction set. The Java run-time stack is replaced by two stacks, one for non-pointer and the other for pointer data. Further, the data structures enable efficient interaction between native methods, Java methods represented by byte code, and Java methods translated by a JIT compiler to native machine language. Performance tradeoffs are biased to give favorable treatment to execution of JIT-translated methods.

Drawing Description Text (2):
FIG. 1 illustrates the organization of thread memory, with each thread comprised of a C stack, and pointer stack, and a non-pointer stack, and each stack represented by multiple stack segments

Drawing Description Text (3):
FIG. 2 illustrates the header information attached to each dynamically allocated memory object for purposes of performing garbage collection. These header fields consist of Scan-List, Indirect-Pointer, Activity-Pointer, Signature-Pointer, and optional Finalize-Link pointers.

Drawing Description Text (6):
FIG. 5 illustrates the appearance of the pointer and non-pointer stack activation

frames immediately before calling and immediately following entry into the body of a Java method. The stacks are assumed to grow downward. In preparation for the call, arguments are pushed onto the stack. Within the called method, the frame pointer (fp) is adjusted to point at the memory immediately above the first pushed argument and the stack pointer (sp) is adjusted to make room for local variables to be stored on the stack.

Drawing Description Text (7):
FIG. 6 illustrates the internal organization of the local-variable region of the stack activation frame. This region includes application-declared locals (as declared in byte-code attributes for Java methods and as specified in the parameterization of BuildFrames( ), temporary variables (as might be required to represent the old values of the frame and instruction pointers), a run-time stack (to allow execution of push and pop operations within the method), and space for arguments to be pushed to other methods to be called from this method.

Drawing Description Text (26):
FIG. 25 provides the C declaration of the structure used internal to the PERC implementation to represent a PERC stack of non-pointers.

Drawing Description Text (31):
FIG. 30 provides a C macro definition of the LongJmpo macro, which is a version of the standard C longjmp( ) function specialized for the PERC virtual machine execution environment. Note that this macro makes use of perclongjmp( ) whose implementation is not provided. perclongjmp( ) expects as parameters a representation of the machine's registers including its instruction pointer, the value of the pointer stack pointer, the value of the non-pointer stack pointer, and the return value to be returned to the point of the JIT version of setjmp( ).

Drawing Description Text (32):
FIG. 31 provides a C declaration of the structure used internal to the PERC implementation to represent a PERC stack of pointers.

Drawing Description Text (33):
FIG. 32 illustrates the signature structure used to represent the memory layout of heap-allocted objects. total.sub.-- length is the total number of words comprising the object, excluding the object's header words, but including its signature if the signature happens to be appended to the end of the data. All pointers are assumed to be word aligned within the structure. Use last.sub.-- descriptor to symbolically represent the word offset of the last word within the corresponding object that might contain a pointer. When the garbage collector scans the corresponding object in search of pointers, it looks no further than the word numbered last.sub.-- descriptor. type.sub.-- code comprises a 2-bit type tag in its most significant bits, with the remaining 30 bits representing the value of last.sub.-- descriptor. bitmap is an array of integers with each integer representing 32 words of the corresponding object, so there are a total of ceiling(last.sub.-- descriptor/32) entries in the array. (bitmap[0]&0.times.01), which represents the first word of the corresponding object, has value 1 if and only if the first word is a pointer.

Drawing Description Text (41):
FIG. 40 provides C macros for conversion between integer offsets and actual derived pointer values and for obtaining the actual address of the constant-pool object. These macros are used to improve the efficiency of access to instruction, stack, and constant-pool memory.

Drawing Description Text (45):
FIG. 44 provides C helper macros for use by application code to coordinate with the dispatcher. TendPointerStack( ), used by SaveThreadState( ), rescans the portion of the pointer stack that is bounded below by .sub.-- gc.sub.-- ps.sub.-- low.sub.-- water and above by .sub.-- psp.

Drawing Description Text (56):
FIG. 55 provides C macros for use in maintaining activation frames on the PERC pointer and non-pointer stacks. The StackOverflowCheck( ) macro is executed each time these stacks expand. The AdjustPSPAndZeroOutLocals( ) macro is executed to zero out the new pointers allocated on the PERC pointer stack. The AdjustLowWaterMacro( ) macro executes each time an activation frame is removed from the pointer stack. The low-water mark identifies the lower limit on the range of the pointer stack that has to be scanned when the task is preempted.

Drawing Description Text (65):
FIG. 64 provides the definitions of C macros for manipulation of the PERC pointer stack.

Drawing Description Text (66):
FIG. 65 provides the definitions of C macros for manipulation of the PERC non-pointer stack.

Drawing Description Text (95):
FIG. 94 illustrates the PERC non-pointer stack activation frame for JIT-generated code. Upon entry into the JIT function, the non-pointer stack pointer (npsp) points to the list of incoming arguments, and the return address is stored in the slot "above" the top-of-stack entry. The prologue of JIT-compiled method subtracts a JIT-computed constant from npsp to make room on the non-pointer stack for saved machine registers, local variables, and outgoing arguments.

Drawing Description Text (100):
FIG. 99 illustrates the implementation of the stackOverflow( ) help routine, which is invoked whenever the PERC pointer or non-pointer stacks are close to overflowing.

Detailed Description Text (12):
Methods represented as byte codes are interpreted by the PERC virtual machine. The interpreter, known throughout this invention disclosure as pvm() (for PERC virtual machine), uses three stacks for execution: (1) the traditional C stack, (2) an explicitly managed stack for representation of PERC pointer values, and (3) an explicitly managed stack for representation of PERC non-pointer values. The C stack holds C-declared local variables and run-time state information associated with compiler generated temporaries. The PERC pointer stack holds the pointer arguments passed as inputs to the method, pointer local variables, temporary pointers pushed during expression evaluation, and pointer values pushed as arguments to methods called by the current method. The PERC non-pointer stack holds non-pointer arguments passed as inputs to the method, non-pointer local variables, temporary non-pointer values pushed during expression evaluation, and non-pointer values pushed as arguments to be called by this method. The pointer and non-pointer stack activation frames are illustrated in FIG. 5 and FIG. 6.

Detailed Description Text (14):
Methods that have been translated to native machine code use only two stacks: the PERC pointer stack and the PERC non-pointer stack. The benefit of using only two rather than three stacks is that this reduces the overhead of stack maintenance associated with each method invocation. The activation frames for the two stacks are structured as illustrated in FIG. 94. However, the amount of information stored in the "temporaries" segment of the activation frame differs between JIT-compiled methods and byte-code methods.

Detailed Description Text (20):
The PERC implementation represents every PERC object with a data structure patterned after the templates provided in FIG. 15, FIG. 16, and FIG. 24. In all of these structures, the second field is a pointer to a MethodTable data structure

(see FIG. 23). The PERC execution environment maintains one MethodTable data structure for each defined object type. All instantiated objects of this type point to this shared single copy. The jit.sub.-- interfaces array field of the MethodTable structure has one entry for each virtual method supported by objects of this type. The mapping from method name and signature to index position is defined by the class loader, as described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley. To execute the JIT version of a PERC method using a virtual method lookup, branch to the code represented by jit.sub.-- interfaces[method.sub.-- index]. Normally, the JIT version of the byte code will only be invoked directly from within another JIT-compiled method. If a native or untranslated byte-code method desires to invoke another method using virtual method lookup, the search for the target method generally proceeds differently. First, we find the target object's MethodTable data structure (as above) and then follow the methods pointer to obtain an array of pointers to Method objects. Within the Method object, we consult the access.sub.-- flags field to determine if the target method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be implemented by byte codes. See FIG. 49, FIG. 45, and FIG. 46.

Detailed Description Text (25):
When the method to be invoked is declared as static within the corresponding object (meaning that the method operates on class information rather than manipulating variables associated with a particular instance of the corresponding class), the Java compiler treats this as an invokeStatic method. Execution of static methods is identical to execution of special methods except that there is no implicit pointer to "this" passed as an argument to the called method. See FIG. 47, FIG. 45, and FIG. 46.

Detailed Description Text (27):
When a method is invoked through an interface declaration, the called method's name and signature is stored as part of the calling method's code representation. The compiler ensures that the object to be operated on has a method of the specified name and signature. However, it is not possible to determine prior to run time the index position within the method table that holds the target method. Thus it is necessary to examine the target object's mtable field, which points to the corresponding MethodTable structure. We follow the MethodTable's methods pointer to find an array of pointers to Method structures. And we search this array for a method that matches the desired name and signature. Once found, we invoke this method. We examine the Method object that represents the target procedure and consult its access.sub.-- flags field to determine if the method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be represented as byte code. See FIG. 50, FIG. 51, FIG. 45, and FIG. 46.

Detailed Description Text (30):
Note that native methods and pvm(), which interprets byte-code methods, use the same stack organization. Thus, calling another method from a native method is the same as calling the method from within the pvm() interpreter. In both cases, the caller invokes the callee by passing appropriate parameters to one of several available invocation routines, all of which are written primarily in C. These invocation routines consult internal fields within the Method structure that describes the callee to determine whether the callee is implemented as byte codes, the JIT translation of byte codes, or a native method (See FIG. 46). The invocation routine adjusts the stack and other state information as necessary in order to transfer control to the called method. When the called method returns, the invocation routine restores the stack and other state information to once again support the execution mode of the calling method. To call a byte-code method, the invocation routine saves the offset of the old frame and stack pointers in local C variables, sets up the callee's activation frames (See FIG. 5), and calls pvm(), passing a pointer to the called method's Method structure as the only argument. To

call a native method, the invocation routine saves the offsets of the old stack and frame pointers, sets up the native method's activation frames (See FIG. 5), and calls (*Method.native)(). To call a JIT-translated method, the invocation routine sets up the callee's activation frames (See FIG. 5), pushes the current C frame pointer onto the C stack and then saves the current value of the C stack pointer in the c.sub.-- sp field of the currently executing thread's Thread data structure, copies the current values of the .sub.-- psp and .sub.-- npsp variables into machine registers dedicated to these purposes (effectively making the PERC stacks become the run-time execution stacks), and branches to (*Method.jit.sub.-- interface)(), leaving the return address in the stack slot above the top-of-stack entry on the non-pointer stack. See FIG. 94 for an illustration of the non-pointer stack activation frame as it is organized during execution of JIT code.

Detailed Description Text (35):
From within the implementation of the pvmo and within native methods, the standard protocol for invoking other methods depends on the type of the call. A virtual method invocation vectors to the corresponding code by way of the target object's method table. The object to which the method corresponds is passed implicitly on the run-time stack. To invoke a virtual method, first push a pointer to the target object onto the pointer stack and then push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeVirtual (), passing as arguments pointers to the String objects that represent the class name and the target method's name and signature (See FIG. 49):

Detailed Description Text (36):
void invokeVirtual(String *class.sub.-- name, String *method.sub.-- name.sub.-- and.sub.-- sig); Note that invokeVirtual() must do a string search within the class representation to find the selected method. This is potentially a costly operation and we would prefer to avoid this cost when possible. When byte code is first loaded into our system, we perform this lookup and save the result, represented by a pointer to a Method structure, within the constant pool. Implementers of native methods may design similar optimizations. There are two mechanisms available to implementers of native methods for the purposes of looking up Method objects: findMethod() and getMethodPtr(). Both of these functions return a pointer to the corresponding Method object. With findMethod(), the desired method is described by a pointer to the known Class object and a String pointer to the method's name and signature. With getMethodPtr(), the desired method is described by String representations of the class name and of the method's name and signature. Prototypes for both functions are provided below:

Detailed Description Text (40):
Within the Method structure, information is available which characterizes the number of pointer arguments of this particular method and the offset of this method within the object's method table (see FIG. 22). To invoke a virtual function without incurring the overhead of a string method lookup, use the FastInvokeVirtualo macro, prototyped below (See FIG. 45):

Detailed Description Text (48):
At the API level, invoking an interface is similar to invoking a virtual or non-virtual method. First push a pointer to the target object onto the pointer stack and then push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeinterface(), passing as arguments String objects representing the name of the class and the name and signature of the method within the class, as prototyped below (See FIG. 50):

Detailed Description Text (52):
Note in the above that the purpose of the template argument is to allow FastInvokeInterface to determine the name and signature of the method that it must search for in the object found num.sub.-- ptr.sub.-- args slots from the current top-of-stack pointer on the PERC pointer stack.

Detailed Description Text (54):
A static method is one that makes use only of information that is associated with the corresponding class (rather than instances of the class). When a static method is invoked, there is no "target object" pushed onto the stack. To call a static method, push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeStatic(), passing as arguments String objects representing the name of the class and the name and signature of the method within the class, as prototyped below (See FIG. 47):

Detailed Description Text (61):
2. Copies the register-held psp and npsp registers into global memory locations .sub.-- psp and .sub.-- npsp. Then assigns sp and fp (the machine's stack and frame pointer registers) to reflect the current C-stack context, as represented by .sub.-- current.sub.-- thread.fwdarw.c.sub.-- sp.

Detailed Description Text (62):
3. Copies the return address off the non-pointer stack (See FIG. 94) and saves its value in a slot within the C stack frame.

Detailed Description Text (63):
4. Calculates and assigns values to .sub.-- pfp (pointer stack frame pointer) and .sub.-- npfp (non-pointer stack frame pointer), based on the current values of the corresponding stack pointers and the number of arguments of each type. The stack activation frames are arranged as illustrated in FIG. 5. Additionally, we adjust the pointer and non-pointer stack pointers to make room for the local variables that are required to execute the method, as represented by the max.sub.-- ptr.sub.-- locals and max.sub.-- non.sub.-- ptr locals fields of the corresponding Method structure.

Detailed Description Text (65):
6. For coordination with the garbage collector, we keep track of how high the pointer stack has grown during the current execution time slice. Since the stack grows downward, the high-water mark is represented by the minimum value of .sub.-- psp.

Detailed Description Text (66):
7. Calls pvm(), passing as a C argument a pointer to the Method object that describes the segment of code to be executed.

Detailed Description Text (70):
11. For coordination with the garbage collector, we keep track of how low the pointer stack has shrunk during the current execution time slice. Since the stack grows downward, the low-water mark is represented by the maximum value of .sub.-- pfp.

Detailed Description Text (76):
To support real-time performance, garbage collection runs asynchronously, meaning that the garbage collection thread interleaves with application code in arbitrary order. To support accurate garbage collection, it is necessary for the garbage collector to always be able to distinguish a thread's pointer variables (including stack-allocated variables and variables held in machine registers) from the thread's non-pointer variables.

Detailed Description Text (77):
To require each thread to maintain all pointers in variables that are at all times easily identifiable by the garbage collector imposes too great an overhead on overall performance. Thus, the PERC virtual machine described in this invention disclosure implements the following compromises:

Detailed Description Text (79):
2. Between preemption points, the thread is allowed to hold pointers in variables that may not be visible to the garbage collector. In this disclosure, we characterize such variables as "fast pointers." Fast pointers are typically declared in C as local variables, and may be represented either by machine registers or slots on the C stack.

Detailed Description Text (80):
3. Pointer variables that are visible to the garbage collector are known throughout this disclosure as "slow pointers". Slow pointers are typically represented by locations on the PERC pointer stack and by certain C-declared global variables identified as "root pointers".

Detailed Description Text (81):
4. Immediately following each preemption, the thread must consider all of its fast pointers to be invalid. In preparation for each preemption, the thread must copy the values of essential fast-pointer variables into slow pointers. Following each preemption, essential fast pointers are restored from the values previously stored in slow pointer variables. Note that, while the thread was preempted, a drefragmenting garbage collector might have relocated particular objects, requiring certain pointer values to be modified to reflect the corresponding objects' new locations.

Detailed Description Text (82):
5. Each C function in the virtual machine implementation is identified as either preemptible or non-preemptible. Before calling a preemptible function, the caller must copy all of its essential fast pointers into slow pointers. When the called function returns, the caller must restore the values of these fast-pointer variables by copying from the slow-pointer variables in which their values were previously stored. Throughout this disclosure, we refer to preemptible functions as "slow functions" and to non-preemptible functions as "fast functions."

Detailed Description Text (86):
1. Do not coerce pointers to integers or integers to pointers.

Detailed Description Text (87):
2. Do not perform any pointer arithmetic unless specifically authorized to do so (e.g. special techniques have been enabled to support efficient instruction and stack pointer operations).

Detailed Description Text (88):
3. Do not store tag information (e.g. low-order bits of a word) within memory locations that are identified as pointers to the garbage collector.

Detailed Description Text (89):
4. Do not store pointers to the C static region or to arbitrary derived addresses in locations identified as garbage-collected pointers, except that pointers to objects residing in the ROMized region are allowed. Note: A derived address is a location contained within an object. The garbage collector assumes all pointers refer to the base or beginning address of the referenced object.

Detailed Description Text (91):
6. When declaring fields and variables that point to the C static region, identify such fields as non-pointers insofar as garbage collection is concerned.

Detailed Description Text (92):
7. Pointers to garbage-collected objects cannot be stored in the C static region unless such pointers have been registered as root pointers.

Detailed Description Text (96):

Given that (base) (field.sub.-- expr) is an expression representing a pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, and that field.sub.-- type represents the type of (base) (field.sub.-- expr), return the fast pointer that represents this heap pointer's value. In the process, we may have to "tend" the pointer's value. Optionally, we may overwrite in place the value of (base) (field.sub.-- expr), so this expression should be a C 1-value. Sample usage:

Detailed Description Text (99):
Given that (base) (field.sub.-- expr) is an expression representing a pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, that field.sub.-- type represents the type of (base) (field.sub.-- expr), and that field.sub.-- value is also of type field.sub.-- type, assign field.sub.-- value to (base) (field.sub.-- expr). In the process, we may have to "tend" field.sub.-- value. Note that (base) (field.sub.-- expr) must be a C 1-value. Sample usage: SetHeapPtr(.sub.-- current.sub.-- thread, .fwdarw.current.sub.-- exception, Thread *, Object *, new.sub.-- exception);

Detailed Description Text (101):
Given that (base) (field.sub.-- expr) is an expression representing a non-pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, and that field.sub.-- type represents the type of (base) (field.sub.-- expr), return the non-pointer value that represents this heap location's value. Sample usage:

Detailed Description Text (104):
Given that (base) (field.sub.-- expr) is an expression representing a non-pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, that field.sub.-- type represents the type of (base) (field.sub.-- expr), and that field.sub.-- value is also of type field.sub.-- type, assign field.sub.-- value to (base) (field.sub.-- expr). Note that (base) (field.sub.-- expr) must be a C 1-value. Sample usage:

Detailed Description Text (114):
With certain garbage collection techniques, it is possible that two fast pointer objects refer to the same object even though their pointer values are different. This might occur, for example, if an object is being copied in order to compact live memory and one pointer refers to the original location of the object and the other pointer refers to the new copy of the object. Programmers should use the SameObject() macro to compare fast pointers for equality. This macro returns non-zero if and only if its two pointer arguments refer to the same object.

Detailed Description Text (142):
5.3 Manipulation of the Pointer Stack

Detailed Description Text (143):
Native methods and other C functions run fastest if they avoid frequent copying of values between local variables (stored on the PERC pointer stacks) and C-declared fast pointers. But pointers stored in C-declared variables are not necessarily preserved across preemption of the thread. Thus, it is necessary for the application code to copy from C-declared variables to macro-declared variables before each preemption. The following macros are used to manipulate the pointer stack.

Detailed Description Text (145):
Push the fast pointer fptr onto the pointer stack.

Detailed Description Text (147):
Pop the fast pointer off of the pointer stack.

Detailed Description Text (149):
Return the fast pointer that is currently offset slots away from the top of the stack.

Detailed Description Text (150):
PeekPtr(0) is the top pointer-stack element.

Detailed Description Text (154):
Return as a fast pointer the local pointer at the specified offset. The first pointer argument is at offset 0. The second pointer argument is at offset 1, and so on.

Detailed Description Text (156):
Set the local pointer at the specified offset to the value of fastptr. Offsets are specified as described in the GetLocalFastPtr() macro.

Detailed Description Text (158):
Adjust the pointer stack pointer (psp) by offset entries. If offset is positive, space representing offset pointers is removed from the stack. ShrinkPS should not be used with a negative offset as this might create a situation in which pointers have garbage values.

Detailed Description Text (159):
5.4 Manipulation of the Non-Pointer Stack

Detailed Description Text (160):
The non-pointer stack holds integers, 8-byte long integers, floating point values, and 8-byte double-precision floating point values. The following macros are suggested for manipulation of the non-pointer stack.

Detailed Description Text (162):
Push integer val onto the non-pointer stack.

Detailed Description Text (164):
Push floating point value val onto the non-pointer stack.

Detailed Description Text (166):
Push 8-byte long value val onto the non-pointer stack.

Detailed Description Text (168):
Push 8-byte double precision value val onto the non-pointer stack.

Detailed Description Text (170):
Pop a single integer from the top of non-pointer stack.

Detailed Description Text (172):
Pop a single floating point value from the top of the non-pointer stack.

Detailed Description Text (174):
Pop a single 8-byte long value from the top of the non-pointer stack.

Detailed Description Text (176):
Pop a single 8-byte double precision value from the top of the non-pointer stack.

Detailed Description Text (178):
Given that off words have been pushed onto the non-pointer stack on top of integer item n, return n.

Detailed Description Text (180):
Given that off words have been pushed onto the non-pointer stack on top of floating

point value x, return x.

Detailed Description Text (182):
Given that off words have been pushed onto the non-pointer stack on top of 8-byte long value m, return m.

Detailed Description Text (184):
Given that off words have been pushed onto the non-pointer stack on top of 8-byte double-precision value y, return y.

Detailed Description Text (186):
Given that off words have been pushed onto the non-pointer stack on top of the integer slot representing n, overwrite this slot with val.

Detailed Description Text (188):
Given that off words have been pushed onto the non-pointer stack on top of the floating point slot representing x, overwrite this slot with val.

Detailed Description Text (190):
Given that off words have been pushed onto the non-pointer stack on top of the 8-byte long integer slot representing m, overwrite this slot with val.

Detailed Description Text (192):
Given that off words have been pushed onto the non-pointer stack on top of the 8-byte double precision slot representing y, overwrite this slot with val.

Detailed Description Text (194):
Given that off words precede the integer variable j within the local non-pointer stack activation frame, return the value of j.

Detailed Description Text (196):
Given that off words precede the floating point variable f within the local non-pointer stack activation frame, return the value of f.

Detailed Description Text (198):
Given that off words precede the 8-byte long integer variable 1 within the local non-pointer pointer stack activation frame, return the value of 1.

Detailed Description Text (200):
Given that off words precede the double precision floating point variable x within the local non-pointer stack activation frame, return the value of

Detailed Description Text (203):
Given that off words precede the integer variable j within the local non-pointer stack activation frame, set the value of j to val.

Detailed Description Text (205):
Given that off words precede the floating point variable f within the local non-pointer stack activation frame, set the value of f to val.

Detailed Description Text (207):
Given that off words precede the 8-byte long integer variable 1 within the local non-pointer stack activation frame, set the value of 1 to val.

Detailed Description Text (209):
Given that off words precede the double precision floating point variable x within the local non-pointer stack activation frame, set the value of x to val.

Detailed Description Text (211):
Adjust the non-pointer stack pointer (npsp) by offset entries. If offset is

positive, space representing offset pointers is removed from the stack. If offset is negative, the specified number of stack slots are added to the stack.

Detailed Description Text (213):
Strict partitioning between fast and slow pointers, and requiring all heap memory access to be directed by way of heap access macros imposes a high overhead. Certain data structures are accessed so frequently that the PERC implementation treats them as special cases in order to improve system performance. In particular, the following exceptions are supported:

Detailed Description Text (214):
1. Note that the PERC stacks dedicated to representation of pointer and non-pointer data respectively are heap allocated. According to the protocols described above, every access to PERC stack data should be directed by way of heap access macros. Since stack operations are so frequent, we allow direct access to stack data using traditional C pointer indirection. This depends on the following:

Detailed Description Text (215):
a. The stack pointers are represented by C global variables declared as pointers. Access to stack data uses C pointer indirection, without enforcement of special read or write barriers. (See FIG. 64 and FIG. 65)

Detailed Description Text (216):
b. Each time the task is preempted, the global variables representing the currently executing thread's stack pointers are saved by the run-time dispatcher in the thread's structure representation (See FIG. 53 and FIG. 44). Note the use of the GetSPOffset() macro (See FIG. 40).

Detailed Description Text (217):
c. Each time a task is scheduled for execution, the dispatcher sets the global stack pointer variables to represent the newly dispatched thread's stack pointers (See FIG. 53 and FIG. 44). Note the use of the GetSP() macro (See FIG. 40).

Detailed Description Text (218):
d. During execution of a time slice, the thread's pointer stack is assumed to hold fast pointers. However, when the thread is preempted, the garbage collector needs to see the stack object's contents as slow pointers. When the thread is preempted, the dispatcher scans that portion of the stack that has been modified during the current time slice in order to convert all fast pointers to slow pointers (See FIG. 53 and FIG. 44). We maintain a low-water mark representing a bound on the range of stack memory that has been impacted by execution of the task during its current time slice to reduce the need for redundant stack scanning.

Detailed Description Text (220):
a. The instruction pointer is represented by a fast pointer declared within the implementation of pvm(). Upon entry into pvm(), this variable is initialized using the GetPC() macro, which expects as arguments a pointer to the ByteString object that represents the method's code and the instruction offset within this method's code (See FIG. 40).

Detailed Description Text (222):
c. After being preempted (or after returning from a function that may have been preempted), the instruction pointer is recomputed by using GetPC().

Detailed Description Text (223):
3. During interpretation of byte-code methods, the constant pool is frequently accessed. Rather than incurring the overhead of a standard heap access macro, we obtain a trustworthy C pointer to the constant pool data structure and refer directly to its contents. For this purpose, we use the GetCP() macro (See FIG. 40). C subscripting expressions based on the value returned by GetCP() are considered

valid up to the time at which the thread is next preempted. Following each preemption, the pointer must be recomputed through another application of the GetCP () macro.

Detailed Description Text (235):
2. Native methods: Like the virtual machine, each invocation of a native method must be preceded by the preparation of PERC stack activation frames. The format of the activation frames and the protocol for allocation of local pointers is exactly the same for native methods as for pvm().

Detailed Description Text (238):
a. Before calling a slow procedure, all fast pointers that are considered to be live must be saved on the PERC pointer stack.

Detailed Description Text (239):
b. Pointer arguments can be passed either on the PERC pointer stack or on the C stack (as regular arguments). Any live arguments passed on the C stack must be saved on the PERC pointer stack prior to calling another slow procedure or reaching a voluntary preemption point.

Detailed Description Text (243):
AllocLocalPointers(num.sub.-- ptrs, first.sub.-- local.sub.-- offset, ptr.sub.-- stack.sub.-- growth, non.sub.-- ptr.sub.-- stack.sub.-- growth); num.sub.-- ptrs specifies the number of pointers for which space is to be reserved within the local activation frame. first.sub.-- local.sub.-- offset is an integer variable that is initialized by this macro to represent the location of the first local variable relative to the beginning of this function's activation frame. ptr.sub.-- stack.sub.-- growth and non.sub.-- ptr.sub.-- stack.sub.-- growth represent the maximum additional stack expansion that might take place during execution of the corresponding stack (through push operations and/or the allocation of stack slots for outgoing arguments). The information provided by these last two arguments is used to perform stack overflow checking and to adjust the pointer stack high-water mark.

Detailed Description Text (244):
To find the offset of the top-of-stack entry within an activation frame that has no local pointers, use the following:

Detailed Description Text (249):
num.sub.-- ptr.sub.-- args and num.sub.-- ptr.sub.-- locals represent the number of incoming pointer arguments and the number of local pointer variables respectively. These variables determine the amount by which the pfp and psp pointers must be adjusted in order to establish the pointer stack activation frame. ptr.sub.-- stack.sub.-- growth is the number of additional stack slots (beyond the slots set aside for locals and arguments) required on the pointer stack to support execution of this slow procedure. This variable is used to check for pointer stack overflow, if such a check is desired. num.sub.-- non.sub.-- ptr.sub.-- args, num.sub.-- non.sub.-- ptr.sub.-- locals, and non.sub.-- ptr.sub.-- stack.sub.-- growth serve the same roles with respect to the non-pointer stack as the corresponding pointer stack variables.

Detailed Description Text (253):
DestroyFrames(num.sub.-- pointers, num.sub.-- non.sub.-- pointers)

Detailed Description Text (254):
DestroyFrames() removes all but num pointers words from the pointer stack and all but num.sub.-- non.sub.-- pointers words from the non-pointer stack. Note that a DestroyFrames() invocation must occur on each control-flow path that reaches either the end of the function's body or a return statement. Prior to invocation of the DestroyFrames() macro, the application code should store the return value into the

0.sup.th slot of the corresponding stack frame.

Detailed Description Text (256):
PrepareJavaFrames(). In preparation for calling the pvm(), as is done within the invoke routines (invokeVirtual(), invokeSpecial(), invokeStatic(), and invokeInterface()) and within byte-code stubs, it is necessary to construct the activation frames for the PERC pointer and non-pointer stacks. This is done by executing the PrepareJavaFrames() macro, with parameters similar to what was described above for

Detailed Description Text (262):
PrepareNativeFrames(). In preparation for calling a native method, as is done within the invoke routines (invokeVirtual(), invokeSpecial(), invokeStatic(), and invokeInterface()) and within byte-code stubs, it is necessary to construct the activation frames for the PERC pointer and non-pointer stacks. This is done by executing the PrepareNativeFrames() macro, with parameters similar to what was described above for BuildFrames():

Detailed Description Text (268):
ReclaimFrames(num.sub.-- pointers, num.sub.-- non.sub.-- pointers);

Detailed Description Text (271):
AdjustLowWaterMark. Both ReclaimFrames() and DestroyFrames() make use of the AdjustLowWaterMark() macro, which is defined in FIG. 55. The purpose of this macro is to keep track of the lowest point to which the pointer stack has shrunk during execution of the current time slice. When this task is preempted, all of the pointers between the low-water mark and the current top-of-stack pointer are tended. By tending these pointers at preemption time, it is not necessary to enforce the normal write barrier with each update to the pointer stack.

Detailed Description Text (273):
The PERC Virtual Machine describes the C function that interprets Java byte codes. This C function, illustrated in FIG. 68, is named pvm(). The single argument to pvm () is a pointer to a Method structure, which includes a pointer to the byte-code that represents the method's functionality. Each invocation of pvm() executes only a single method. To call another byte-code method, pvm() recursively calls itself. Note that pvm() is reentrant. When multiple Java threads are executing, each thread executes byte-code methods by invoking pvm() on the thread's run-time stack.

Detailed Description Text (274):
The implementation of pvm() allocates space on the PERC pointer stack for three pointer variables. These pointers, known by the symbolic names pMETHOD, pBYTECODF, and pCONSTANTS, represent pointers to the method's Method structure, the StringOfBytes object representing its byte code, and the constant-pool object representing the method's constant table respectively. During normal execution of pvm(), the values of these variables are stored in the C locals method, bytecode, and cp respectively. Before preemption, and before calling preemptible functions, pvm() copies the contents of these C variables onto the PERC pointer stack. In preparation for executing the byte codes representing a byte-code method, pvm() checks to determine if the method has any exception handlers. If the method is synchronized, the lock will have been obtained by the fastInvokeo routine prior to calling pvm() (see FIG. 46). However, fastInvoke() does not set an exception handler to release the lock if the code is aborted by the raising of an exception. For this reason, pvm() sets an exception handler if the method is synchronized, so that it can release the lock before rethrowing the exception to the surrounding context.

Detailed Description Text (283):
The IADD instruction removes the top two elements from the non-pointer stack, both of which are known to represent integers, adds these two integer values, and stores

the sum onto the top of the same stack. This is illustrated in FIG. 69.

Detailed Description Text (284):
The AASTORE instruction removes from the pointer stack a reference to an array and a reference to an object to be inserted into an array, and removes from the non-pointer stack an integer index representing the position within the array that is to be overwritten with the new value. This instruction makes sure that the array subscript is within bounds and makes sure that the value to be inserted into the array is of the proper type. Then it stores the reference into the array at the specified index position, as illustrated in FIG. 70.

Detailed Description Text (285):
The FCMPL instruction removes the top two elements from the non-pointer stack, both of which are known to represent floating point numbers, compares these two values, and stores an integer representing the result of comparison onto the same stack. The result is encoded as 0 if the two numbers are equal, -1 if the first is less than the second, and 1 if the first is greater than the second. The implementation of FCMPL, is illustrated in FIG. 71.

Detailed Description Text (286):
The IFEQ instruction (See FIG. 72) branches to the byte-code instruction obtained by adding the two-byte signed quantity which is part of the instruction encoding to the current value of the program pointer if the top of the non-pointer stack, which is known to represent an integer, holds the value 0. Note that the PVMPreemptionPointo macro appears before the break statement. pvm() allows itself to be preempted at this point. In general, pvm() considers each byte-code instruction which may cause control branching to be a preemption point. This guarantees that there is at least one preemption point in each byte-code loop.

Detailed Description Text (287):
The JSR instruction (See FIG. 73) jumps to a subroutine by branching to the byte-code instruction obtained by adding the two-byte signed quantity which is part of the instruction encoding to the current value of the program counter and pushing the return address onto the non-pointer stack. Note that the return address is represented as the integer offset within the current method's byte code rather than an actual pointer. This is because the garbage collector does not deal well with pointers that refer to internal addresses within objects rather than to the objects' starting addresses. Note also that the JSR instruction also invokes the PVMPremptionPoint() macro.

Detailed Description Text (289):
The TABLESWITCH instruction (See FIG. 75) is used to efficiently implement switch statements in which most of the various cases are represented by consecutive integers. The immediate-mode operands of this instruction are encoded as (1) padding to align the next operand at an address that is a multiple of 4 bytes, (2) a low integer value representing the first integer in the range of cases, (3) a high integer value representing the last integer in the range of cases, (4) the integer representing the byte-code offset of the code that represents the default case, and (5) (high+1-low) integers representing the byte-code offsets of the code that implements each of the cases. This instruction removes the top entry, which is known to be an integer, from the non-pointer stack and uses this value to index into the branch table in order to compute the address of the next instruction to be executed. Note that TABLESWITCH invokes the PVMPreemptionPoint() macro.

Detailed Description Text (290):
The LOOKUPSWITCH instruction (See FIG. 76) is used to implement switch statements in which the cases are not consecutive integers. The immediate-mode operands of this instruction are encoded as (1) padding to align the next operand on an address that is a multiple of 4 bytes, (2) an integer representing the total number of cases, (3) the integer representing the byte-code offset of the code that

represents the default case, and (4) pairs of key values combined with instruction offsets for each of the cases identified in field number 2. This instruction removes the top entry from the non-pointer stack, which is known to be an integer, and searches for this value among the cases represented in its encoding. Note that LOOKUPSWITCH invokes the PVMPreemptionPoint() macro.

Detailed Description Text (291):
The IRETURN instruction (See FIG. 77) is used to return an integer from the currently executing method. This instruction pops the integer value to be returned from the top of the non-pointer stack and stores the integer value into the 0th slot of the non-pointer stack's current activation frame. Then it breaks out of the interpreter loop by using a goto statement.

Detailed Description Text (292):
The GETSTATIC.sub.-- QNP8 instruction (See FIG. 78) gets an 8-bit non-pointer value from the static area of the class corresponding to the field that is stored in the constant-pool table at the offset specified by this instruction's one-byte immediate-mode operand. The value fetched from the static field is pushed onto the non-pointer stack.

Detailed Description Text (293):
The PUTFIELLD.sub.-- Q instruction stores a value (provided on one of the PERC stacks) into the specified field of a particular object. A pointer to the object that contains the field is passed on the pointer stack. The two-byte immediate operand of this instruction indexes into the constant pool to find a 4-byte integer value. This integer value encodes the offset of the field within the object as the least significant 29 bits, an encoding of the number of bits to be updated if the field is not a pointer in the next two most significant bits, and a flag distinguishing pointer fields in the most significant bit. The implementation of this instruction is illustrated in FIG. 79.

Detailed Description Text (294):
The INVOKEVIRTUAL.sub.-- FQ instruction (See FIG. 80) invokes a virtual function. The method-table index is encoded as the first immediate-mode byte operand and the number of pointer arguments is encoded as the second immediate-mode byte operand. Note that most of the work associated with invoking the virtual method is performed by the FastInvokeVirtual() macro, which is illustrated in FIG. 45. Note also that pvm() saves and restores its state surrounding the method invocation.

Detailed Description Text (296):
Invocation of interfaces is performed by the INVOKEINTERFACE.sub.-- Q instruction (See FIG. 83). Invoking interfaces is inherently more complicated than the other forms of invocation because the method table of the target object must be searched for a method with a matching name and signature. It is not generally possible to map the name and signature to an integer index prior to execution of the instruction. The immediate-mode operands to this instruction are (1) a one-byte index into the constant pool table to obtain a pointer to a method that has the desired name and signature, (2) a one-byte operand representing the number of pointer arguments passed to the interface method, and (3) a one-byte guess as to the offset within the target object's method table at which the target method will be found. See the definition of the FastInvokeInterface() macro in FIG. 45.

Detailed Description Text (297):
The NEW.sub.-- Q instruction (See FIG. 84) allocates a new object. This instruction takes a two-byte immediate-mode operand, which is an index into the constant pool. The corresponding entry within the constant pool is a pointer to the Class object (See FIG. 16) that describes the type of the object to be allocated. The newly allocated object is pushed onto the pointer stack.

Detailed Description Text (298):

The NEWARRAY instruction (See FIG. 85) allocates a new array of non-pointer data. The type of the non-pointer data is encoded as a one-byte immediate-mode operand to the instruction. The size of the array is passed as an integer on the non-pointer stack. The newly allocated array is pushed onto the pointer stack.

Detailed Description Text (299):
The ANEWARRAY.sub.-- Q instruction (See FIG. 86) allocates a new array of pointers. The type of the array entry is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the corresponding Class object. The size of the array is passed as an integer on the non-pointer stack. The newly allocated array is pushed onto the pointer stack.

Detailed Description Text (300):
The ATHROW instruction (See FIG. 87) throws the exception object that is on the top of the pointer stack. Note that this instruction causes control to longjmp out of the current pvm() activation. Where the exception is caught, the stacks will be truncated to the appropriate heights. Thus, it is not necessary to pop the thrown exception.

Detailed Description Text (301):
The CHECKCAST.sub.-- Q instruction (See FIG. 88) ensures that the top pointer stack element is of the appropriate type, where appropriate type is defined to mean that the type of the stack element is derived from the "desired" type. If it is not, this instruction throws an exception. The desired type is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the Class object that represents the desired type. Note that the NULL value is considered to match any reference type. If the top pointer stack value is of the appropriate type, the value is left on top of the pointer stack.

Detailed Description Text (302):
The INSTANCEOF.sub.-- Q instruction (See FIG. 89) removes the top pointer stack element and checks to see if it is of the appropriate type, where appropriate type is defined to mean that the type of the stack element is the "desired" type. If it is, this instruction pushes a 1 onto the non-pointer stack. If it isn't, this instruction pushes a 0 onto the non-pointer stack. The NULL value is considered to be of the appropriate type. The desired type is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the Class object that represents the desired type.

Detailed Description Text (303):
The MONITORENTER instruction (See FIG. 90) removes the object reference on the top of the pointer stack and arranges to apply a semaphore-like lock on that object. If the object is already locked by another thread, the current thread is put to sleep until the object becomes unlocked. Note that the pvm()'s state is saved and restored surrounding the call to the enterMonitor() function, because that call may result in preemption of this thread. Note that if the entry on the top of the pointer stack is NULL, this instruction throws an exception. The MONITOREXIT instruction (See FIG. 91) removes the object reference on the top of the pointer stack and arranges to remove its semaphore-like lock on that object. If the object has been locked multiple times by this thread, this instruction simply decrements the count on how many times this object has been locked rather than removing the lock. As with the MONITORENTER instruction, pvm( )'s state is saved and restored surrounding the call to the exitMonitor() function and this instruction throws an exception of the top of the pointer stack is NULL.

Detailed Description Text (307):
1. The current values of the pointer stack and frame pointers.

Detailed Description Text (308):
2. The current values of the non-pointer stack and frame pointers.

Detailed Description Text (309):
3. The explicitly saved value of the C stack pointer, for situations in which the exception handling context is established from within JIT-compiled code. While JIT-compiled code is executing, the C stack is not used, and the value of the C stack pointer is held in a special field of the corresponding Thread object.

Detailed Description Text (310):
4. A pointer to the surrounding exception handling context. The special supplemental information fields are stored within the PERCEnvironment data structure, which is illustrated in FIG. 26.

Detailed Description Text (314):
In the PERC implementation, every object has a HashLock pointer field, which is initialized to NULL. When either a lock or a hash value is needed for the object, a HashLock object (see FIG. 20) is allocated and initialized, and the HashLock pointer field is made to refer to this HashLock object. Note that each HashLock object has the following fields:

Detailed Description Text (316):
2. The u field is a union which can represent either a pointer to another HashLock object (in case this HashLock object is currently residing on a free list), or a pointer to the thread that owns this semaphore if the lock is currently set, or NULL if this object is not currently on a free list and the lock is not currently set.

Detailed Description Text (319):
Obtaining a hash value. When application code desires to obtain the hash value of a particular object, it invokes the native hashCode() method. This method consults the object's lock field. If this field is NULL, this method allocates a HashLock object, initializes its hash.sub.-- value field to the next available hash value, and initializes the object's lock pointer to refer to the newly alocated HashLock object. Then it returns the contents of the hash.sub.-- value field. If the lock field is non-NULL, hashCode() consults the hash.sub.-- value field of the corresponding HashLock object to determine whether a hash value has already been assigned. If this field has value 0, hashCode() overwrites the field with the next available hash value. Otherwise, the hash value has already been assigned. In all cases, the last step of hashCode() is to return the value of the hash.sub.-- value field.

Detailed Description Text (332):
Each thread is represented by a Thread object which includes instance variables representing the critical state information associated with the thread (See FIG. 36). One of the instance variables points to the jump buffer (PERCEnvironment) of the currently active exception handler context. Each thread maintains three stacks, one to represent C activation frames, one to represent non-pointer PERC arguments and local variables, and a third to represent PERC pointer arguments and local variables. JIT-generated code uses only the two PERC stacks. While executing JIT-generated code, the C stack pointer is stored in a Thread field so that the machine's stack-pointer register can refer to the non-pointer PERC stack. We desire to allocate small stacks so as to conserve memory. This is especially important for applications comprised of large numbers of threads. For reliability, we provide stack overflow checking and, in some cases, the capability of expanding stacks on the fly as necessary.

Detailed Description Text (333):
We say that the C stack segments contain no pointers, but this is not entirely true. Since the C activation frame contains return addresses, the stack contains pointers to code. And since the activation frame includes saved registers, it probably contains the saved values of frame and stack pointers. To avoid the

complications and efficiency hits that would be associated with the handling of these pointers by a relocating garbage collector, we require stack segments to be non-moving, except for one exception which is discussed below.

Detailed Description Text (334):
The C stack may also contain pointers to heap objects which were saved in registers or local variables within particular activation frames. The usage protocol requires that such variables be treated as dead insofar as the garbage collector is concerned.

Detailed Description Text (335):
All three kinds of stacks are represented by multiple stack segments. In general, each run-time stack is allowed to expand on the fly as necessary. Expansion occurs whenever a stack overflow is detected. Expansion consists of allocating a new stack segment, copying that portion of the original stack segment that is necessary to establish an execution context on the new stack segment (the incoming parameters, for example), adjusting links to represent the addition of the new stack segment and setting the corresponding stack pointer(s) to their new values. The data structures are illustrated in FIG. 1.

Detailed Description Text (337):
The sample implementation characterized by this invention disclosure uses operating system provided stack overflow checking and stack expansion for the C stack, and uses explicit software overflow checks for the PERC pointer and non-pointer stacks. The C stack overflow checking is performed using memory management hardware.

Detailed Description Text (348):
1. The watchdog task is written entirely in C. Thus, it does not make use of the PERC stack and frame pointer variables.

Detailed Description Text (354):
1. Registers as root pointers .sub.-- gc.sub.-- thread and .sub.-- dispatcher.sub.-- thread. These static variables identify the Thread objects that govern the garbage collection thread and the real-time dispatcher thread respectively.

Detailed Description Text (385):
11. Sets the global .sub.-- current.sub.-- thread pointer to refer to the dispatcher task.

Detailed Description Text (389):
When a new thread is created, the system allocates a C stack, a PERC non-pointer stack, and a PFRC pointer stack. The size of the C stack is determined as a run-time option (specified on the command line if the virtual machine is running in a traditional desktop computing environment). The size of the PERC pointer and non-pointer stacks is specified by compile-time macro definitions, defined to equal 1024 words per stack.

Detailed Description Text (392):
A compile-time constant represents a threshold test for proximity to the PERC stack overflow conditions (See P.sub.-- SAFETY.sub.-- PADDING and NP.sub.-- SAFETY.sub.-- PADDING in FIG. 55). Within the BuildFrames(), PrepareJavaFrames(), and AllocLocalPointers() macros, we test whether the current stack pointer is within this threshold of overflowing the corresponding stack. If so, we immediately create a new stack frame for execution of this procedure by:

Detailed Description Text (395):
3. Recursively calling this same procedure by way of a trampoline function which is responsible for restoring the stack to "normal" upon return from the recursive invocation. Note that certain code sequences may result in "thrashing" of the stacks in the sense that every time we call a particular procedure, we grow the

stack and every time the procedure returns, we shrink it. If we end up in a loop that repeatedly calls this procedure, we will find ourselves spending too much time managing the stack growth. A solution to this problem is to keep track of how frequently particular stacks need to be expanded. If a particular stack is expanded too frequently, then we will consider it worth our while to expand the stack contiguously. Contiguous expansion of the stack consists of creating a single larger stack segment that is large enough to represent multiple C stack segments and copying the first C stack segment onto this new stack. While copying the stack data, it is necessary to adjust stack pointers that refer to the stack. Primarily, this consists of the frame pointer information that might be stored on the C stack.

Detailed Description Text (397):
So-called fast pointers refer directly to the corresponding memory objects using traditional C syntax. Fast pointers are stored on the traditional C stack or in machine registers. They are not scanned by the garbage collector. Thus, it is very important to make sure that garbage collection occurs at times that are coordinated with execution of application threads. (If the garbage collector were to relocate an object "while" an application thread was accessing that object by way of a fast pointer, the application thread would become confused.) Each application thread is responsible for periodically checking whether the system desires to preempt it. The following macro serves this purpose:

Detailed Description Text (405):
Note that each time we preempt a task, we must be prepared to save and restore all of the fast pointers that are currently in use. However, in cases where a particular pointer variable is known to have been saved to the stack already, and has not been modified since it was last saved to the stack, it is possible to omit the save operation. The purpose of copying fast pointers into "local" pointer variable slots is to make them visible to the garbage collector. After the task has been preempted, the application task's fast pointers may no longer be valid. Thus, the application task must restore the fast-pointer variables by copying their updated values from the local pointer variables.

Detailed Description Text (408):
2. Rescanning all of the PERC stack pointer's data found between the stack's low-water mark and the current stack pointer. Then set the low-water stack mark to refer to the current stack activation frame. (The last of these two operations may be postponed until after this thread is resumed.)

Detailed Description Text (417):
In the first case, the protocol described immediately above ensures that local variables are in a consistent state at the moment the task is preempted. To handle the second case, we require that any C code in the run-time system that calls a non-fast function consider all of its fast pointers to have been invalidated by invocation of the non-fast function. Further, we require that the invocation of blocking system calls be surrounded by the PrepareBlockCall() and ResumeAfterBlockCall() macros, as shown below:

Detailed Description Text (431):
Native libraries are implemented according to a protocol that allows references to dynamic objects to be automatically updated whenever the dynamic object is relocated by the garbage collector. However, if these native libraries call system routines which do not follow the native-library protocols, then the system routines are likely to become confused when the corresponding objects are moved. To avoid this problem, programmers who need to pass heap pointers to system libraries must make a stable copy of the heap object and pass a pointer to the stable copy. The stable copy should be allocated on the C stack, as a local variable. If necessary, upon return from the system library, the contents of the stable copy should be copied back into the heap. Note that on uniprocessor systems a non-portable

performance optimization to this strategy is possible when invoking system libraries that are known not to block if thread preemption is under PERC's control. In particular, we can pass the system library a pointer to the dynamic object and be assured that the dynamic object will not be relocated (since the garbage collector will not be allowed to run) during execution of the system library routine.

Detailed Description Text (439):
8.3 Global Pointer Variables (Roots)

Detailed Description Text (440):
All global root pointers must be registered so that they can be identified by the garbage collector at the start of each garbage collection pass. These root pointers are independently registered using the RegisterRooto macro, prototyped below. Each root pointer must be registered before its first use.

Detailed Description Text (451):
Each heap-allocated object must be identified so that the garbage collector can determine which of its fields contain pointers. The standard technique for identifying pointers within heap objects is to provide a signature for each object. The signature pointer occupies a particular word of each object's header (See FIG. 2).

Detailed Description Text (452):
The signature structure is illustrated in FIG. 32. The total.sub.-- length field counts the total number of words in the corresponding object. The type.sub.-- code field comprises two kinds of information: a four-bit code identifying the kind of object and a twenty-eight-bit integer that identifies the word offset of the last pointer contained within this object. If there are no pointers contained within the object, the word offset has value zero. The most significant bit of type.sub.-- code is set to indicate that the corresponding object needs to be finalized. The next three most-significant bits encode the kind of object, as represented by the preprocessor constants in FIG. 33. These special constants are manipulated using the macros provided in FIG. 34.

Detailed Description Text (453):
Within the signature structure, bitmap is an array of bits with one bit representing each word of the corresponding object. The bit has value zero if the corresponding word is a non-pointer, and value one if the corresponding word is a pointer. The first word of the object is represented by (bitmap[0]& 0.times.01). The second word is represented by (bitmap[0]& 0.times.02). The thirty-third word is represented by (bitmap[1]& 0.times.01), and so forth. Bits are provided only up to the word offset of the last pointer. Note that multiple heap-allocated objects may share the same statically allocated signature structure.

Detailed Description Text (473):
The special preprocessor converts the signature macro to the following declaration: static int.sub.-- sig1234[ ]={5, Record .vertline.5, 0.times.01f,}; static struct Signature *.sub.-- sigClassFile=(struct Signature *).sub.-- sig1234; The codes used to identify fields within a structure are the same as the primitive C types: char, short, int, long, float, double. Note that we need not distinguish unsigned values. The ptr keyword represents pointers (the garbage collector does not need to know the type of the object pointed to).

Detailed Description Text (495):
Every PERC object begins with two special fields representing the object's lock and method tables respectively. See FIG. 23 for the declaration of MethodTable. The method table's first field is a pointer to the corresponding Class object. The second field is a pointer to an array of pointers to Method objects. The third field is a pointer to the JIT-code implementation of the first method, followed by

a pointer to the JIT-code implementation of the second method, and so on. The pointers to JIT-code implementations may actually be pointers only to stub procedures that interface JIT code to byte-code or native-code methods.

Detailed Description Text (496):
Allocation routines. When allocating memory from within a native method, the programmer provides to the allocation routine the address of a signature rather than simply the size of the object to be allocated. The Signature pointer passed to each allocate routine must point to a statically allocated Signature structure. The implementation of the PERC virtual machine allocates a static signature for each class loaded. Once this static signature has been created, all subsequent instantiations of this class share access to this signature.

Detailed Description Text (499):
Note that every allocated object is tagged according to which real-time activity allocated it. This is necessary in order to allow the run-time system to enforce memory allocation budgets for each activity. Allocations performed by traditional Java applications that are not executing as part of a real-time activity are identified by a null-valued Activity pointer. All of the allocate routines consult the Thread referenced by.sub.-- current.sub.-- thread to determine which Activity the current thread belongs to.

Detailed Description Text (500):
In some cases, such as when a dynamically allocated object contains union fields that contain pointers only some of the time, it is necessary to allocate a private copy of the signature along with the actual object. To minimize allocation overhead, both the signature and the data are allocated as a single contiguous region of memory using the following allocation routine, which assumes that its sp argument points to static memory:

Detailed Description Text (506):
In some situations, it is necessary to allocate a region of memory within which particular fields will contain both pointer and non-pointer data. Such an object is allocated using the allocUnionArray() routine, prototyped below:

Detailed Description Text (508):
This routine allocates an object with the specified number of words and an accompanying signature within which all tags are initially set to indicate that fields contain non-pointers.

Detailed Description Text (509):
If the type of a particular word of this object must be changed at some later time to a pointer, its type tag is modified by using the setSigPtrTag() routine:

Detailed Description Text (511):
This routine sets the tag for the object at word.sub.-- offset positions from the start of obj to indicate that the corresponding word contains a pointer. As a side effect, this routine overwrites the corresponding word with NULL. If at some later time it is necessary to change the word from a pointer to a non-pointer, use the clrSigPtrTag() routine:

Detailed Description Text (515):
Slow versions of each routine are prototyped below. These slow functions pass pointer parameters and return pointer results on the C stack. Prior to preemption, the routine saves relevant pointers to slow pointer variables set aside on the PERC pointer stack for this purpose.

Detailed Description Text (521):
String and substring data is special in that we may have arrays of bytes that are shared by multiple overlapping strings. The bytes themselves are represented in a

block of memory known to the garbage collector as a String. The programmer represents each string using a String object. FIG. 7 shows string objects x and y, representing the strings "embedded" and "bed" respectively. The value field of each string object is a pointer to the actual string data. The offset field is the offset, measured in bytes, of the start of the string within the corresponding StringData buffer. The count field is the number of bytes in the string. Note that count represents bytes, even though Unicode strings might require two bytes to represent each character.

Detailed Description Text (530):
Slow versions of each of the routines described above are prototyped below. These slow functions pass pointer parameters and return pointer results on the C stack. Prior to preemption, the routine saves relevant pointers to slow pointer variables set aside on the PERC pointer stack for this purpose.

Detailed Description Text (543):
For objects residing in from-space which have been scheduled for copying into to-space, the Scan List field is overwritten with a pointer to the to-space copy. Otherwise, the Scan List field holds NULL.

Detailed Description Text (545):
b. The Indirect Pointer refers to the currently valid copy of the data that corresponds to this object. For objects in the mark and sweep region, this pointer always points to the object itself. For objects in to- and from-space, the pointer points to whichever version of the object currently represents the object's contents.

Detailed Description Text (546):
c. Activity Pointer points to the real-time activity object that was responsible for allocation of this object or has the NULL value if this object was not allocated by a real-time activity. When this object's memory is reclaimed, that real-time activity's memory allocation budget will be increased. Furthermore, if this object needs to be finalized when the garbage collector endeavors to collect it, the object will be placed on a list of this real-time activity's objects which are awaiting finalization. To distinguish objects that need to be finalized, the 0.times.01 bit (FINAL.sub.-- LINK) and the 0.times.02 bit (FINAL.sub.-- OBJ) of the Activity Pointer field are set when a finalizable object is allocated.

Detailed Description Text (547):
d. Signature Pointer points to a structure that represents the internal organization of the PERC data within the object. For objects requiring finalization, the Finalize Link field is not represented in the signature.

Detailed Description Text (548):
4. Free segments are doubly linked. The Indirect Pointer field is used as a forward link and the Signature Pointer field is used as the backward link. The size of the free segment, in words, is stored in the Activity Pointer field, representing an integer. Note that objects residing on a free list are distinguished by the special SCAN.sub.-- FREE value stored in their Scan List field.

Detailed Description Text (553):
In Java, programmers can specify an action to be performed when objects of certain types are reclaimed by the garbage collector. These actions are specified by including a non-empty finalize method in the class definition. Such objects are said to be finalizable. When a finalizable object is allocated, the two low order bits of the Activity Pointer are set to indicate that the object is finalizable. The 0.times.01 bit, known symbolically as FINAL.sub.-- LINK, signifies that this object has an extra Finalize Link field appended to the end of it. The 0.times.02 bit, known symbolically as FINAL.sub.-- OBJ, signifies that this object needs to be finalized. After the object has been finalized once, its FINAL.sub.-- OBJ is

cleared, but its FINAL.sub.-- LIINK bit remains on throughout the object's lifetime.

Detailed Description Text (554):
See FIG. 3 for an illustration of how finalization lists are organized. In this figure, Finalizees is a root pointer. This pointer refers to a list of finalization-list headers. There is one such list for each of the currently executing real-time activities, and there is one other list that represents all of the objects allocated by non-real-time activities. These lists are linked through the Activity Pointer field of the objects waiting to be finalized.

Detailed Description Text (555):
The run-time system includes a background finalizer thread which takes responsibility for incrementally executing the finalizers associated with all of the objects reachable from the Finalizees root pointer. Following execution of the finalizer method, the finalizes object is removed from the finalizes list and its Activity Pointer field is overwritten with a reference to the corresponding Activity object. Furthermore, we clear the FINAL.sub.-- OBJ so we don't finalize it again. Optionally, each real-time activity may take responsibility for timely finalization of its own finalizee objects. Typically, this is done within an ongoing real-time thread that is part of the activity's workload.

Detailed Description Text (556):
When an Activity object is first allocated, its pointer to the corresponding finalizee list head object is initialized to null. Later, when objects requiring finalization are encountered, a finalizee list head object is allocated and the Activity object's finalizee list head pointer is overwritten with a pointer to this object. Each time the activity's finalizee list becomes empty, we destroy the corresponding finalizee list head object, removing it from the Finalizees list, and overwrite the corresponding pointer within the Activity object with NULL.

Detailed Description Text (557):
The Finalize Link field is only present in objects that have finalization code. Throughout their lifetimes, all such objects have the FINAL.sub.-- LINK bit of the Activity Pointer field set at all times (and no objects that lack a Finalize Link field ever have this bit set). When first allocated, each finalizable object is linked through the Finalize Link field onto a single shared list (called the finalizable list) that represents all finalizable objects. When an object is recognized as ready for finalization, it is removed from the finalizable list and placed onto a finalizee list threaded through the Activity Pointer field.

Detailed Description Text (560):
1. Heap memory that has already been examined by the garbage collector must not be corrupted by writing into such heap objects pointers that have not yet been processed by the garbage collector. Otherwise, it might be possible for a pointer to escape scrutiny of the garbage collector. As a result, the referenced object might be treated as garbage and accidentally reclaimed. To avoid this problem, we impose a write barrier whenever pointers are written into the heap. (See the SetHeapPointer() macro in FIG. 41):

Detailed Description Text (561):
a. If the pointer to be written to memory refers to from-space, replace the pointer with the appropriate to-space address. Note that this may require that we set aside memory in to-space to hold the copy of the referenced from-space object.

Detailed Description Text (562):
b. If the pointer to be written to memory refers to a mark-and-sweep object that has not yet been marked, mark the object by placing it on the scan list.

Detailed Description Text (563):

2. We do not impose a read barrier. This means that pointers fetched from the internal fields of heap objects may refer to from-space objects or to mark-and-sweep objects that have not yet been marked. In case a pointer refers to a from-space object that has already been copied into to-space or to a to-space object that has not yet been copied into to-space, all references to heap object are indirected through the Indirection Pointer. (See FIG. 41)

Detailed Description Text (564):
3. Any objects that are newly allocated from the mark-and-sweep region have their Scan List pointer initialized to NULL. Thus, newly allocated objects will survive the current garbage collection pass only if pointers to these objects are written into the live heap.

Detailed Description Text (571):
2. Tend each root pointer. This consists of:

Detailed Description Text (572):
a. If the pointer refers to from-space, allocate space for a copy of this object in to-space and make the to-space copy's Indirect Pointer refer to the from-space object. Set the root pointer to refer to the to-space copy. Set the from-space copy's Scan List pointer to refer to the to-space copy.

Detailed Description Text (573):
b. Otherwise, if the pointer refers to the mark-and-sweep region or the to-space region and the referenced object has not yet been marked, mark the object. Marking consists of placing the object on the scan list. Each increment of garbage collection effort consists of the following:

Detailed Description Text (583):
7. Else, do a flip operation and restart the garbage collector. To-space and from-space are organized as illustrated in FIG. 4. In this illustration, live objects A, B, and C are being copied into to-space out of from-space. Objects B and C have been copied and object A is on the copy queue waiting to be copied. The arrows indicate the values of the Indirect Pointer fields in each of the invalid object copies. Memory to the right of the New pointer consists of objects that have been allocated during this pass of the garbage collector. Memory to the left of B' represents objects that were copied to to-space during the previous pass of the garbage collector. Garbage collection of the copy region consists of the following:

Detailed Description Text (585):
a. Atomically copy the object at position Relocated and update the from-space version of the object so that its Indirect Pointer refers to the to-space copy of the object.

Detailed Description Text (586):
b. As the object is being copied, tend any pointers that it might contain.

Detailed Description Text (587):
Additionally, tend the Activity Pointer field after masking out its two least significant bits and update the Signature Pointer if the signature is contained within this object. Scanning of the mark-and-sweep region consists of the following:

Detailed Description Text (589):
a. Scan the object at the head of the list. Scanning consists of tending each pointer contained within the object. Note that the scanner must scan the Activity Pointer field (after masking out the two least significant bits). A special technique is used to scan pointer stack objects. When pointer stack objects are scanned, the garbage collector consults the corresponding Thread object to

determine the current height of the pointer stack. Rather than scan the entire object, the garbage collector only scans that portion of the stack object that is currently being used.

Detailed Description Text (590):
b. Make the scan-list pointer refer to the next object on the scan list.

Detailed Description Text (591):
Scanning of PERC pointer stacks is special in the sense that only the portion of the stack that is live is scanned. Memory within the object that is above the top-of-stack pointer is ignored. In order to support this capability, PERC pointer stacks refer to their corresponding Thread object, enabling the garbage collector to consult the thread's top-of-stack pointer before scanning the stack object.

Detailed Description Text (595):
We remove the object from the finalizable list and place it (the newly created to-space copy if the object was originally found in from-space) onto a temporary holding list of finalizees threaded through the Finalize Link field. In order to support the remove operation, the scanning process maintains at all times a pointer to the preceding object on the finalizable list. Additionally, we mark this object by placing it on the scan list if the object resides in the mark-and-sweep region.

Detailed Description Text (598):
3. Wait for the scanning and copying process to complete. It is not necessary to rescan the root pointers because all of the objects now being scanned and copied are considered to be dead insofar as the application code is concerned. Thus, there is no possible way for a pointer to one of these "dead" objects to find its way into a root pointer.

Detailed Description Text (599):
4. Now, process the holding list of finalizees that was created in step 1, linking each finalizes onto the appropriate activity's finalizes list (or onto the Orphaned Finalizees list). This list is threaded through the Activity pointer field of the object's header. At this time, overwrite the object's Finalize Link field with NULL. If the activity to which an object corresponds does not currently have a finalizes list, it will be necessary in this step to allocate and initialize the finalizee list head. (See FIG. 3)

Detailed Description Text (606):
The final step is to zero out the old from-space so that future allocations from this region can be assumed to contain only zeros. Simply walk through memory from low to high address and overwrite each word with a zero. For each object encountered in from-space, we ask whether it was copied into to-space (by examining its Indirect Pointer). If it was not copied, we check to see if it is a HashLock object with a hash value that needs to be reclaimed. If so, we reclaim the hash value as described above, except that a new HashCache object may need to be allocated to represent the recycled hash value if there are no available slots in the existing list of recycled hash values. We allocate this HashCache object using the standard heap-memory allocator. Otherwise, we update the corresponding activity's tally that represents the total amount of this activity's previously allocated memory that has been garbage collected.

Detailed Description Text (608):
The standard model for execution of Java byte-code programs assumes an execution model comprised of a single stack. Furthermore, the Java byte codes are designed to support dynamic loading and linking. This requires the use of symbolic references to external symbols. Resolving these symbolic references is a fairly costly operation which should not be performed each time an external reference is accessed. Instead, the PERC virtual machine replaces symbolic references with more efficient integer index and direct pointer references when the code is loaded.

Detailed Description Text (609):
In order to achieve good performance, the PERC virtual machine does not check for type correctness of arguments each time it executes a byte-code instruction. Rather, it assumes that the supplied arguments are of the appropriate type. Since byte-code programs may be downloaded from remote computer systems, some of which are not necessarily trustworthy, it is necessary for the PERC virtual machine to scrutinize the byte-code program for type correctness before it begins to execute. The process of guaranteeing that all of the operands supplied to each byte-code instruction are of the appropriate type is known as byte code verification. Once the types of each operation are known, it is possible to perform certain code transformations. Some of these transformations are designed simply to improve performance. In other cases, the transformations are needed to comply with the special requirements of the PERC virtual machine's stack protocols. For example, Java's dup2 byte code duplicates the top two elements on the Java stack. Byte-code verification determines the types of the top two stack elements. If both are of type pointer, the class loader replaces this byte code with a special instruction named dup2.sub.-- 11, which duplicates the top two elements of the pointer stack. If the two stack arguments are both non-pointer values, the PERC class loader replaces this byte code with the dup2.sub.-- 00 instruction, which duplicates the top two elements of the non-pointer stack. If one of dup's stack arguments is a pointer and the other is a non-pointer (in either order), the PERC class loader replaces dup with dup2.sub.-- 10, which duplicates the top element on each stack. A complete list of all the transformations that are performed by the byte code loader is provided in the remainder of this section.

Detailed Description Text (616):
2. A list of pointers to the basic block objects that may branch to this block. We call these blocks the predecessors.

Detailed Description Text (617):
3. A list of pointers to the basic block objects that this block may branch to. We call these blocks the successors.

Detailed Description Text (633):
item found on the specified one-byte indexed position within the constant pool table onto the stack. If this item is an object pointer, we need to push the pointer value onto the pointer stack. If this item is not a pointer, we push its value onto the non-pointer stack. We use code 18 to represent ldc1.sub.-- np, which loads a non-pointer constant onto the non-pointer stack. We use code 255 to represent ldc1.sub.-- p, which loads a pointer constant onto the pointer stack. ldc2. This operation is represented by code 19. This instruction pushes the item found on the specified two-byte indexed position within the constant pool table onto the stack. If this item is an object pointer, we need to push its value onto the pointer stack. If this item is not a pointer, we push its value onto the non-pointer stack. We use code 19 to represent ldc2.sub.-- np, which loads a non-pointer constant onto the non-pointer stack. We use code 254 to represent ldc2.sub.-- p, which loads a pointer constant onto the pointer stack.

Detailed Description Text (635):
1. putfield.sub.-- q encoded as 181: We replace the constant-pool entry with an integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less than 64 Kbytes and can thus be represented in the 16-bit immediate operand

representing an unsigned integer quantity.

Detailed Description Text (636):
2. putfield.sub.-- qnp8 encoded as 253: We replace the two-byte immediate operand with the offset of the 8-bit non-pointer field within the corresponding object.

Detailed Description Text (637):
3. putfield.sub.-- qnp16 encoded as 252: We replace the two-byte immediate operand with the offset of the 16-bit non-pointer field within the corresponding object.

Detailed Description Text (638):
4. putfield.sub.-- qnp32 encoded as 251: We replace the two-byte immediate operand with the offset of the 32-bit non-pointer field within the corresponding object.

Detailed Description Text (639):
5. putfield.sub.-- qnp64 encoded as 250: We replace the two-byte immediate operand with the offset of the 64-bit non-pointer field within the corresponding object.

Detailed Description Text (640):
6. putfield.sub.-- qp encoded as 249: We replace the two-byte immediate operand with the offset of the 32-bit pointer field within the corresponding object.

Detailed Description Text (642):
1. getfield.sub.-- q encoded as 180: We replace the constant-pool entry with a 32-bit integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less than 64 Kbytes and can thus be represented in the 16-bit immediate operand representing an unsigned integer quantity.

Detailed Description Text (643):
2. getfield.sub.-- qnp8 encoded as 248: We replace the two-byte immediate operand with the offset of the 8-bit non-pointer field within the corresponding object.

Detailed Description Text (644):
3. getfield.sub.-- qnp16 encoded as 247: We replace the two-byte immediate operand with the offset of the 16-bit non-pointer field within the corresponding object.

Detailed Description Text (645):
4. getfield.sub.-- qnp32 encoded as 246: We replace the two-byte immediate operand with the offset of the 32-bit non-pointer field within the corresponding object.

Detailed Description Text (646):
5. getfield.sub.-- qnp64 encoded as 245: We replace the two-byte immediate operand with the offset of the 64-bit non-pointer field within the corresponding object.

Detailed Description Text (647):
6. getfield.sub.-- qp encoded as 244: We replace the two-byte immediate operand with the offset of the 32-bit pointer field within the corresponding object.

Detailed Description Text (648):
Putstatic. This operation is represented by code 179. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a

pointer to the Field structure that describes the field to be updated. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (651):
3. putstatic.sub.-- qnp32 encoded as 241 if the field is a 32-bit non-pointer.

Detailed Description Text (652):
4. putstatic.sub.-- qnp64 encoded as 240 if the field is a 64-bit non-pointer.

Detailed Description Text (653):
5. putstatic.sub.-- qp encoded as 239 if the field is a 32-bit pointer.

Detailed Description Text (654):
Getstatic. This operation is represented by code 178. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a pointer to the Field structure that describes the field to be fetched. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (657):
3. getstatic.sub.-- qnp32 encoded as 236 if the field is a 32-bit non-pointer.

Detailed Description Text (658):
4. getstatic.sub.-- qnp64 encoded as 235 if the field is a 64-bit non-pointer.

Detailed Description Text (659):
5. getstatic.sub.-- qp encoded as 234 if the field is a 32-bit pointer.

Detailed Description Text (660):
Anewarray. This operation is represented by code 189. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool. When resolved, the selected constant must be a class. The result of this operation is a newly allocated array of pointers to the specified class. The loader replaces this instruction with anewarray.sub.-- q, which is also encoded as operation 189. This instruction differs from anewarray in that it does not need to resolve the constant entry. Rather, it assumes that the specified slot of the constant pool holds a pointer directly to the corresponding class object.

Detailed Description Text (661):
Multianewarray. This operation is represented by code 197. It takes two immediate-mode byte operands to represent a 16-bit constant pool index and a third immediate-mode byte operand to represent the number of dimensions in the array to be allocated. The index position is handled the same as for anewarray. The loader replaces this instruction with multianewarray.sub.-- q, which is encoded as operation 197. This instruction differs from multianewarray in that it does not need to resolve the constant entry. Rather, it assumes that the specified slot of the constant pool holds a pointer directly to the corresponding class object.

Detailed Description Text (662):
Invokevirtual. This operation is represented by code 182. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method signature, including its name. If the method-table index of the corresponding method is greater than 255 or if the number of pointer arguments is greater than 255, the loader replaces this instruction with invokevirtual.sub.-- q, encoded as operation 182. Otherwise, the

loader replaces this instruction with invokevirtual.sub.-- fq, encoded as operation 233.

Detailed Description Text (663):
With the invokevirtual.sub.-- fq instruction, the first immediate-mode byte operand represents the method table index and the second immediate-mode byte operand represents the number of pointer arguments.

Detailed Description Text (664):
With the invokevirtual.sub.-- q instruction, the two immediate-mode operands represent the same 16-bit index into the constant pool table as with the original invokevirtual instruction. However, this entry within the constant pool table is overwritten with a pointer to the Method structure that describes this method. (Note that both invokevirtual and invokespecial may share access to this same entry in the constant pool. In fact, there is no difference between the implementations of invokespecial.sub.-- q and invokestatic.sub.-- q in certain frameworks.)

Detailed Description Text (665):
invokespecial. This operation is represented by code 183. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method signature, including its name. This instruction is replaced with invokespecial.sub.-- q, encoded as 183. With the invokespecial.sub.-- q instruction, the selected constant pool cntry is replaced with a pointer to the Method structure that describes this method. (Note that both invokevirtual and invokespecial may share access to this same entry in the constant pool.)

Detailed Description Text (666):
invokestatic. This operation is represented by code 184. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method's class and signature, including its name. The loader replaces this instruction with invokestatic.sub.-- q, encoded as 184. The distinction of invokestatic.sub.-- q is that the selected constant pool entry is a pointer to the Method structure that describes this method.

Detailed Description Text (667):
Invokeinterface. This operation is represented by code 185. The instruction takes a 2-byte constant pool index, a one-byte representation of the number of arguments, and a one-byte reserved quantity as immediate-mode operands. The corresponding constant-pool entry represents the method's signature. The loader replaces this instruction with invokeinterface.sub.-- q, encoded as 185. The distinction of invokeinterface.sub.-- q is that the constant pool entry is overwritten with a pointer to a Method structure that represents the name and signature of the interface method and the reserved operand is overwritten with a guess suggesting the "most likely" slot at which the invoked object's method table is likely to match the invoked interface. If this slot does not match, this instruction searches the object's method table for the first method that does match. On each execution of invokeinterface.sub.-- q, the guess field is overwritten with the slot that matched on the previous execution of this instruction.

Detailed Description Text (669):
New. This operation is represented by code 187. The instruction takes a 2-byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with new.sub.-- q, also encoded as 187. The distinction of new.sub.-- q is that the constant pool entry is replaced with a pointer to the resolved class object.

Detailed Description Text (670):
Checkcast. This operation is represented by code 192. The instruction takes a 2-

byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with checkcast.sub.-- q, also encoded as 192. The distinction of checkcast.sub.-- q is that the constant pool entry is replaced with a pointer to the resolved class object.

Detailed Description Text (671):
Instanceof. This operation is represented by code 193. The instruction takes a 2-byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with instanceof.sub.-- q, also encoded as 193. The distinction of instanceof.sub.-- q is that the constant pool entry is known to have been replaced with a pointer to the resolved class object.

Detailed Description Text (673):
The standard Java byte code assumes that all local variables and all push and pop operations refer to a single shared stack. Offsets for local variables are all calculated based on this assumption. Our implementation maintains two stacks, one for non-pointers and another for pointers. Pointer local variables are stored on the pointer stack. And non-pointer locals are stored on the non-pointer stack. Thus, our byte-code loader has to remap the offsets for all local variable operations. The affected instructions are: iload, iload.sub.-- <n>, lload, lload.sub.-- <n>, fload, fload.sub.-- <n>, dload, dload.sub.-- <n>, aload, aload.sub.-- <n>, istore, istore.sub.-- <n>, lstore, lstore.sub.-- <n>, fstore, fstore.sub.-- <n>, dstore, dstore.sub.-- <n>, astore, astore.sub.-- <n>, iinc.

Detailed Description Text (676):
We want to make sure that operations that manipulate the stack are properly configured to differentiate between the pointer stack and the non-pointer stack.

Detailed Description Text (677):
Pop. This operation is represented by code 87. It removes the top item from the stack. We usc code 87 to represent pop.sub.-- 0, which pops from the non-pointer stack, and code 232 to represent pop.sub.-- 1, which pops from the pointer stack.

Detailed Description Text (678):
pop2. This operation is represented by code 88. It removes the top two items from the stack. We use code 88 to represent pop2.sub.-- 00, which pops two values from the non-pointer stack, code 231 to represent pop2.sub.-- 10 which pops one value from each stack, and code 230 to represent pop2.sub.-- 11, which pops two values from the pointer stack.

Detailed Description Text (679):
Dup. This operation is represented by code 89. It duplicates the top stack item. We use code 89 to represent dup.sub.-- 0, which duplicates the top non-pointer stack entry, and code 229 to represent dup.sub.-- 1, which duplicates the top pointer stack entry.

Detailed Description Text (680):
dup2. This operation is represented by code 92. It duplicates the top two stack items. We use code 92 to represent dup.sub.2.sub.-- 00, which duplicates the top two non-pointer stack entries, code 228 to represent dup2.sub.-- 10, which duplicates the top entry on each stack, and code 227 to represent dup2.sub.-- 11, which duplicates the top two pointer stack entries.

Detailed Description Text (681):
dup .times.1. This operation is represented by code 90. It duplicates the top stack item, shifts the top two stack items up one position on the stack, and inserts the duplicated top stack item into the newly vacated stack position. Note that the translation of this instruction depends on the types of the top two stack values at

the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 01 represents the condition in which the top stack element is a pointer and the next entry is a non-pointer. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (682):
00: We use code 90 to represent dup.sub.-- .times.1.sub.-- 00, which does its manipulations entirely on the non-pointer stack.

Detailed Description Text (683):
01: Reuse code 229 to represent dup.sub.-- 1, which duplicates only a single pointer value (this is the appropriate action to perform when the top stack element is a pointer, and the second element is a non-pointer).

Detailed Description Text (684):
10: Reuse code 89 to represent dup.sub.-- 0, which duplicates only a single non-pointer value (this is the appropriate action to perform when the top stack element is a non-pointer and the second element is a pointer).

Detailed Description Text (685):
11: Use code 226 to represent dup.sub.-- .times.1.sub.-- 11, which does all of its manipulations on the pointer stack.

Detailed Description Text (686):
dup .times.2. This operation, encoded as 91, duplicates the top stack entry, shifts the top three stack entries up one stack position, and inserts the duplicated stack entry into the newly vacated stack position. Note that the translation of this instruction depends on the types of the top three stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 001 represents the condition in which the top stack element is a pointer and the next two entries are non-pointers. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (687):
000: We use code 91 to represent dup.sub.-- .times.2.sub.-- 000, which does its manipulations entirely on the non-pointer stack.

Detailed Description Text (688):
001: Reuse code 229 to represent dup.sub.-- 1, which duplicates only a single pointer value (this is the appropriate action to perform when the top stack element is a pointer, and the next two elements are non-pointers).

Detailed Description Text (689):
010: Reuse code 90 to represent dup.sub.-- .times.1.sub.-- 00, which duplicates the non-pointer value and inserts it into the appropriate position on the non-pointer stack.

Detailed Description Text (690):
011: Reuse code 226 to represent dup.sub.-- .times.1.sub.-- 11, which duplicates the pointer value and inserts it into the appropriate position on the pointer stack.

Detailed Description Text (691):
100: Reuse code 90 to represent dup.sub.-- .times.1.sub.-- 00, which duplicates the

non-pointer value and inserts it into the appropriate position on the non-pointer stack.

Detailed Description Text (692):
101: Reuse code 226 to represent dup.sub.-- .times.1.sub.-- 11, which duplicates the pointer value and inserts it into the appropriate position on the pointer stack.

Detailed Description Text (693):
110: Reuse code 89 to represent dup.sub.-- 0, which duplicates only a single non-pointer value (this is the appropriate action to perform when the top stack element is a non-pointer and the second element is a pointer).

Detailed Description Text (694):
111: We use code 225 to represent dup.sub.-- '2.sub.-- 111, which does its manipulations entirely on the pointer stack.

Detailed Description Text (695):
dup2 .times.1. This operation, encoded as 93, duplicates the top two stack entries, shifts the top three stack entries up two stack positions, and inserts the duplicated stack entries into the newly vacated stack slots. Note that the translation of this instruction depends on the types of the top three stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 001 represents the condition in which the top stack element is a pointer and the next two entries are non-pointers. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (696):
000: We use code 93 to represent dup2.sub.-- .times.1.sub.-- 000, which does its manipulations entirely on the non-pointer stack.

Detailed Description Text (699):
011: Reuse code 227 to represent dup2.sub.-- 11, which duplicates the top two values on the pointer stack.

Detailed Description Text (700):
100: Reuse code 92 to represent dup2.sub.-- 00, which duplicates the top two values on the non-pointer stack.

Detailed Description Text (703):
111: We use code 222 to represent dup2.sub.-- .times.1.sub.-- 111, which does its manipulations entirely on the pointer stack.

Detailed Description Text (704):
dup2 .times.2. This operation is represented by code 94. It duplicates the top two stack items, shifts the top four stack items up two positions on the stack, and inserts the duplicated stack items into the newly vacated stack positions. Note that the translation of this instruction depends on the types of the top four stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 0001 represents the condition in which the top stack element is a pointer and the next three are non-pointers. Each combination of four binary digit type codes represents a decimal number. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (705):
0000: We use code 94 to represent dup2.sub.-- .times.2.sub.-- 0000, which does all its manipulations on the non-pointer stack.

Detailed Description Text (708):
0011: We reuse the code for dup2.sub.-- 11 (which is the right thing to do if the top two stack elements are pointers, and the next two are non-pointers).

Detailed Description Text (709):
0100: We reuse code for dup2.sub.-- .times.1.sub.-- 000. This instruction duplicates the top two entries on the non-pointer stack, shifts the top three entries of the non-pointer stack up two positions, and inserts the duplicated values into the vacated stack slots.

Detailed Description Text (712):
0111: We reuse the code for dup2.sub.-- .times.1.sub.-- 111. This instruction duplicates the top two entries of the pointer stack, shifts the top three values of the pointer stack up two positions on the stack, and inserts the duplicated pointer values into the vacated stack position.

Detailed Description Text (720):
1111: We use code 218 to represent dup2.sub.-- .times.2.sub.-- 1111. This instruction duplicates the top two entries on the pointer stack, shifts the top four entries on the pointer stack up two positions, and inserts the duplicated pointer values into the newly vacated pointer slots.

Detailed Description Text (722):
00: We use code 95 to represent swap.sub.-- 00, which exchanges the top two values on the non-pointer stack.

Detailed Description Text (725):
11: We use code 217 to represent swap.sub.-- 11, which exchanges the top two values on the pointer stack.

Detailed Description Text (727):
JIT-generated methods usc only the PERC pointer and non-pointcr stacks. All pointer information is stored on the pointer stack and all non-pointer information is stored on the non-pointer stack. The non-pointer activation frame is illustrated in FIG. 94. The pointer activation frame is identical except that there is no return address stored in the pointer activation frame.

Detailed Description Text (728):
Within a JIT-generated method, all local variables, including incoming and outgoing arguments are referenced at fixed offsets from the register that represents the corresponding stack pointer. There is no need for a frame pointer because the stack pointer remains constant throughout execution of the method.

Detailed Description Text (729):
Note that the JIT method's prologue subtracts a constant value from the stack pointer and the JIT method's epilogue adds the same constant value to the stack pointer.

Detailed Description Text (730):
When JIT-compiled methods invoke byte-code or native-codc methods, the corresponding byte-code stub sets up the frame and stack pointers necessary for execution of the corresponding C routines. Additionally, the return address is removed from the non-pointer stack and stored temporarily in a C local variable within the stub procedure.

Detailed Description Text (731):

Within a JIT-compiled method, machine registers are partitioned so that certain registers are known to only contain base pointers and other machine registers are known to only contain non-pointers. An additional class of registers may contain derived pointers which refer to the internal fields of particular objects. Each derived-pointer register is always paired with a base-pointer register which is known to identify the starting address of the corresponding object. Otherwise, the derived-pointer register holds the NULL value.

Detailed Description Text (741):
2. All Indirect Pointers are initialized to refer to the object itself. This enables standard heap-access macros to work correctly when referring to ROM objects.

Detailed Description Text (744):
a. For ROM preloading, the representation of a class data structure is simply a template which will be copied into RAM when the system actually executes. Since the executing application code must be able to modify the class's static variables, the class representation's static variable pointer points to a signature (represented within the load file) which is used by the bootstrap "loader" to allocate the class's static variable structure.

Detailed Description Paragraph Table (3):
Field Name Field Size Description
Object.sub.-- Size 32 bits This represents the total number of words in the object region (Object.sub.-- Region). Relocatable.sub.-- M (Object.sub.-- Siz This field maintains 1 bit for each word of ap e/32) words the object region. The bit is on if the (rounded corresponding word holds a non-null up) pointer and is off otherwise. All non-null pointers are assumed to point within the object region. The first word of the object region (Object.sub.-- Region) is represented by bit 0x01 of the first word of the Relocatable.sub.-- Map. Class.sub.-- Table 32 bits This field represents the offset within the object region (Object.sub.-- Region) of the table that represents all of the classes defined by this object. Object.sub.-- Region Object.sub.-- Size This represents the ROM memory image. words Each object in this memory region is provided with a standard garbage collection header (See Figure 2), including a Scan List pointer, Indirect Pointer, Activity Pointer, and Signature Pointer. In the ROM image, all pointers, including the pointers stored within object headers, are represented by offsets relative to the beginning of the Object.sub.-- Region. All objects are initialized to belong to the 0 Activity.

CLAIMS:

1. A real-time virtual machine method (RTVMM) for implementing real-time systems and activities, the RTVMM comprising the steps:

implementing an O-OPL program that can run on computer systems of different designs, an O-OPL program being based on an object-oriented programming language (O-OPL) comprising object type declarations called classes, each class definition describing the variables that are associated with each object of the corresponding class and all of the operations called methods that can be applied to instantiated objects of the specified type, a "method" being a term of art describing the unit of procedural abstraction in an object-oriented programming system, an O-OPL program comprising one or more threads wherein the run-time stack for each thread is organized so as to allow accurate identification of type-tagged pointers contained on the stack without requiring type tag information to be updated each time the stack's content changes, the O-OPL being an extension of a high-level language (HLL) exemplified by Java, HLL being an extension of a low-level language (LLL) exemplified by C and C++, a thread being a term of art for an independently-executing task, an O-OPL program being represented at run time by either O-OPL byte

codes or by native machine codes.

2. The RTVMM of claim 1 wherein an O-OPL program utilizes a pointer stack and a non-pointer stack.

12. The RTVMM of claim 1 wherein one of the implemented threads is a garbage collection thread that operates asynchronously thereby resulting in the garbage collection thread being interleaved with other threads in arbitrary order, objects subject to garbage collection being either finalizable or non-finalizable, a finalizable object being subject to an action that is performed when the memory space allocated to the finalizable object is reclaimed by the garbage collection thread, the finalizing action being specified by including a non-empty finalizer method in the class definition, the garbage collection thread being able to distinguish a thread's pointer variables from the thread's non-pointer variables, preemption of a thread being allowed only if the thread is in a state identified as a preemption point, a thread being allowed to hold pointers in variables between preemption points that may not be visible to the garbage collection thread, pointer variables that may not be visible to the garbage collection thread being called fast pointers, pointer variables that are visible to the garbage collection thread being called slow pointers, each LLL, function being identified as either preemptible or non-preemptible.

13. The RTVMM of claim 12 wherein the implementing step comprises the steps:

causing the values of essential fast pointers to be copied into slow pointers immediately prior to a preemption point of a preemptible thread. (5.0/4)

causing the values of essential fast pointers to be restored after preemption by causing the values of the slow pointers to be copied to the locations where the values of the fast pointers were previously stored.

14. The RTVMM of claim 12 wherein the implementing step comprises the steps:

causing the values of all of the essential fast pointers of a preemptible LLL function to be copied into slow pointers prior to calling the prcemptible LLL function;

causing the values of the essential fast pointers to be restored when the called preemptible LLL function returns by causing the values of the slow pointers to be copied to the locations where the values of the fast pointers were previously stored.

17. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that returns the value of a fast pointer in the heap given the identity of the pointer and its type.

18. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that assigns a value from a fast pointer in heap memory given the identity of the pointer, its type, and the value.

21. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing direct access to stack data using LLL pointer indirection.

22. The RTVMM of claim 21 wherein the implementing step comprises the step:

representing stack pointers by LLL global variables declared as pointers.

25. The RTVMM of claim 24 wherein the implementing step comprises the step:

including an "activity pointer" field for each object in memory, the "activity pointer" identifying the activity that was responsible for allocating the object, the "activity pointer" field containing a "null" value if the object was not allocated by a real-time activity.

26. The RTVMM of claim 25 wherein the implementing step comprises the step:

maintaining a free pool of space segments for to-space and for each mark-and-sweep sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "activity pointer" field to specify the size of a free space segment.

27. The RTVMM of claim 24 wherein the implementing step comprises the step:

including a "signature pointer" field for each object in memory, the "signature pointer" field containing a pointer to a structure that represents the internal organization of the O-OPL data within the object.

28. The RTVMM of claim 27 wherein the implementing step comprises the steps:

maintaining a free pool of space segments for to-space and for each mark-and-sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "signature pointer" field to be used as a backward link to the preceding segment.

30. The RTVMM of claim 24 wherein the implementing step comprises the step:

including a "scan list" field for each object in memory, the "scan list" field distinguishing marked and unmarked objects residing in a mark-and-sweep space but not on a free list, the "scan list" field for each object in a mark-and-sweep space having a "scan clear" value at the beginning of a garbage collection cycle, an object recognized as being a live object being placed on a list of recognized live objects, the "scan list" field for an object on the list of recognized live objects having either a "scan end" value denoting the last object on the list of recognized live objects or a value identifying the next object on the list of recognized live objects, the "scan list" field for an object residing on a free list within a mark-and-sweep space or to- space having the "scan free" value, the "scan list" field for an object residing in from-space which has been scheduled for copying into to-space being a pointer to the to-space copy, the "scan list" field otherwise being assigned the "scan clear" value, the "scan list" field for an object residing in to-space having the "scan clear" value at the beginning of a garbage collection cycle, a to-space object recognized as live during garbage collection being placed on a list of recognized live objects, the "scan list" field for a to-space object on the list of recognized live objects having a value identifying the next object on the list of recognized live objects, the "scan list" field for each object queued for copying into to-space having the "scan end" value denoting that the object is live.

34. The RTVMM of claim 24 wherein the implementing step comprises the steps:

causing a "finalize link" bit and a "finalize object" bit in an "activity pointer"

field of a finalizable object to be set when space is allocated to the finalizable object, the "finalize link" bit being set indicating that the object has a "finalize link" field appended to the object, the "finalize object" bit being set indicating that the object needs to be finalized;

causing the "finalize object" bit to be cleared when a finalizable object has been finalized.

35. The RTVMM of claim 24 wherein a pointer is to be written into memory, the implementing step comprising the steps: causing the pointer to an object in from-space to be replaced by a pointer to the object's new address in to-space;

causing an object in mark-and-sweep space to which the pointer points to be marked if the object has not yet been marked.

38. The RTVMM of claim 24 wherein the implementing step comprises the steps:

maintaining a list of root pointers to live objects;

causing space for a copy of an object in to-space to be allocated if a root pointer to the object refers to from-space;

causing the from-space address of the object to be written in an "indirect pointer" field of the object's allocated space in to-space;

causing the root pointer to be replaced with the address of the object in to-space;

causing the to-space address of the object to be written into a "scan list" field of the object in from-space.

39. The RTVMM of claim 24 wherein the implementing step comprises the steps:

maintaining a list of root pointers to live objects;

causing an object to be marked if the root pointer to the object refers to a mark-and-sweep space or to-space and the object has not yet been marked, marking consisting of placing the object on a scan list.

43. The RTVMM of claim 40 wherein the transfer of objects needing finalization on the list of finalizable objects to the finalizee list has been completed, the implementing step comprising the steps:

causing the objects on the copy list to be copied to to-space;

causing the objects on the scan list to be scanned, scanning consisting of tending each pointer contained within an object, tending being a term of art describing the garbage collection process of (1) examining a pointer and, if the object has not already been recognized as live, arranging for the referenced object to be subsequently scanned by placing the object on a scan list if it resides in a mark-and-sweep space or in to-space or by arranging for the object to be copied into to-space if it resides in from-space and (2) updating the pointer to refer to the object's new location if it has been queued for copying into to-space.

45. The RTVMM of claim 44 wherein an activity's finalizee list is implemented by placing in an "activity pointer" field of a finalizee the address of the next finalizes on the activity's finalizee list.

46. The RTVMM of claim 44 wherein after transferring a finalizee on the finalizee list to the appropriate activity's finalizee list or onto an orphaned finalizee

list, the implementing step comprises the step:

causing a "finalize link" bit in an "activity pointer" field of the object corresponding to the finalizee to be cleared, a cleared "finalize link" bit indicating that the object is no longer on the list of finalizable objects.

51. The RTVMM of claim 12 wherein the implementing step comprises the step:

designating portions of memory as a to-space, from-space, and zero or more mark-and-sweep spaces;

including an "indirect pointer" field for each object in memory, the "indirect pointer" field containing a pointer to the location of the currently valid copy of the data that corresponds to the object, the pointer pointing to the object itself for objects in a mark-and-sweep space, the pointer pointing to the location of the object that currently represents the object's contents for objects in to-space and from-space.

52. The RTVMM of claim 51 wherein the implementing step comprises the steps:

maintaining a free pool of space segments for to-space and for each mark-and-sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "indirect pointer" field to be used as a forward link to the succeeding segment.

53. The RTVMM of claim 12 wherein the implementing step comprises the step:

including an "activity pointer" field for each object in memory, the "activity pointer" identifying the real-time activity object that was responsible for allocation of the object, the "activity pointer" field containing a "null" value if the object was not allocated by a real-time activity;

maintaining a finalizees list of objects waiting to be finalized for each real-time activity, the objects on the finalizees list being linked through the "activity pointer" field;

maintaining a list of the headers of the finalizees lists, the pointer "finalizees" being a root pointer to the headers list.

54. The RTVMM of claim 53 wherein the implementing step comprises the step:

implementing a finalizer thread that operates in the background and is

responsible for incrementally executing the finalizer methods associated with finalizee objects reachable from the "finalizees" pointer.

55. The RTVMM of claim 54 wherein the finalizer thread comprises the steps:

causing a finalizer method associated with a finalizee object to be executed;

causing the finalizee object to be removed from the associated finalizee list;

causing the "activity pointer" field of the finalizee object to be overwritten with a reference to the allocating object;

causing a "finalize object" bit in the "activity pointer" field of the finalizee

object to be cleared indicating that the object has been finalized.

56. The RTVMM of claim 53 wherein the implementing step comprises the step:

implementing a finalizer thread that is part of a real-time activity and is responsible for incrementally executing the finalizer methods associated with finalizee objects associated with the real-time activity and reachable from the "finalizees" pointer.

57. The RTVMM of claim 56 wherein the finalizer thread comprises the steps:

causing a finalizer method associated with a finalizee object to be executed;

causing the finalizee object to be removed from the associated finalizee list;

causing the "activity pointer" field of the finalizee object to be overwritten with a reference to the allocating object;

causing a "finalize object" bit in the "activity pointer" field of the finalizee object to be cleared indicating that the object has been finalized.

58. The RTVMM of claim 53 wherein the implementing step of claim 1 comprises the steps:

causing memory space to be allocated to a finalizee list head object when an object associated with a particular activity and requiring finalization is encountered;

causing a finalizee list head pointer associated with the activity to be overwritten with a pointer to the finalizee list head object;

causing the finalizee list head object to be destroyed when the finalizee list becomes empty and overwriting the finalizee list head pointer with the "null" value.

72. The RTVMM of claim 1 wherein the implementing step comprises the step:

causing symbolic references to be replaced with integer indices and direct pointer references when a program is loaded into a computer system.

77. The RTVMM of claim 1 wherein an O-OPL byte-code loader is used to load an HLL byte-code program into a computer system, the implementing step comprises the step:

causing each byte code to be examined when a class is loaded to determine whether it operates on pointer or non-pointer data;

causing pointers to be pushed onto and popped from a pointer stack;

causing non-pointers to bc pushed onto and popped from a non-pointer stack.

79. The RTVMM of claim 1 wherein the implementing step comprises the step:

utilizing only O-OPL pointer and non-pointer stacks in executing methods compiled by a JIT compiler, JIT standing for "just in time" and denoting a process for translating HLL, byte codes to native machine language codes on the fly, just in time for its execution, the translation of byte codes to native codes being a form of JIT compiling.

80. The RTVMM of claim 79 wherein a method compiled by a JIT compiler invokes a byte-code or nativc-code method, the implementing step comprises the step:

causing the frame and stack pointers necessary for the execution of the corresponding LLL routines to be set up;

causing the return address to be removed from the non-pointer stack and stored temporarily in an LLL local variable.

84. The RTVMM of claim 1 wherein each thread maintains its own versions of global variables "pointer stack pointer" (psp), "pointer stack frame pointer" (pfp), "non-pointer stack pointer" (npsp) and "non-pointer stack frame pointer" (npfp), the implementing step comprising the steps:

creating a thread called a thread dispatcher, the thread dispatcher saving psp, pfp, npsp, and npfp into the state variables of an executing thread when the executing thread is preempted, the preempted thread restoring these state variables when the preempted thread resumes execution.

87. The RTVMM of claim 85 wherein the implementing step relating to the load file includes the step:

causing the "indirect pointer" field of each object to refer to itself.

First Hit Fwd Refs

End of Result Set

☐ Generate Collection Print

L28: Entry 1 of 1 File: USPT Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A
TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Abstract Text (1):
The invention is a method for use in executing portable virtual machine computer programs under real-time constraints. The invention includes a method for implementing a single abstract virtual machine execution stack with multiple independent stacks in order to improve the efficiency of distinguishing memory pointers from non-pointers. Further, the invention includes a method for rewriting certain of the virtual machine instructions into a new instruction set that more efficiently manipulates the multiple stacks. Additionally, using the multiple-stack technique to identify pointers on the run-time stack, the invention includes a method for performing efficient defragmenting real-time garbage collection using a mostly stationary technique. The invention also includes a method for efficiently mixing a combination of byte-code, native, and JIT-translated methods in the implementation of a particular task, where byte-code methods are represented in the instruction set of the virtual machine, native methods are written in a language like C and represented by native machine code, and JIT-translated methods result from automatic translation of byte-code methods into the native machine code of the host machine. Also included in the invention is a method to implement a real-time task dispatcher that supports arbitrary numbers of real-time task priorities given an underlying real-time operating system that supports at least three task priority levels. Finally, the invention includes a method to analyze and preconfigure virtual memory programs so that they can be stored in ROM memory prior to program.

Brief Summary Text (4):
Java (a trademark of Sun Microsystems, Inc.) is an object-oriented programming language with syntax derived from C and C++. However, Java's designers chose not to pursue full compatibility with C and C++ because they preferred to eliminate from these languages what they considered to be troublesome features. In particular, Java does not support enumerated constants, pointer arithmetic, traditional functions, structures and unions, multiple inheritance, goto statements, operator overloading, and preprocessor directives. In their place, Java requires all constant identifiers, functions (methods), and structures to be encapsulated within

Brief Summary Text (14):
Accurate Garbage Collection, as the term is used in this invention disclosure, describes garbage collection techniques in which the garbage collector has complete knowledge of which memory locations hold pointers and which don't. This knowledge is necessary in order to defragment memory.

Brief Summary Text (16):
Conservative Garbage Collection, as the term is used in this invention disclosure, describes garbage collection techniques in which the garbage collector makes conservative estimates of which memory locations hold pointers. Conservatively, the garbage collector assumes that any memory location holding a valid pointer value (a

legal memory address) contains a pointer. Fully conservative garbage collectors cannot defragment memory. However, partially conservative garbage collectors (in which some pointers are accurately identified) can partially defragment memory.

Brief Summary Text (21):
Fast Pointer is a term specific to this invention disclosure which describes pointers that are implemented using the fastest possible techniques available on a particular computer system. Fast pointers are "normal" pointers as they would be implemented by a typical compiler for the C language.

Brief Summary Text (32):
Pointer is a term of art describing a value held within computer memory or computer registers for the purpose of identifying some other location in memory. The value "points" to a memory cell.

Brief Summary Text (36):
Root Pointer is a term of art describing a pointer residing outside the heap which may point to an object residing within the heap. The garbage collector considers all objects reachable through some chain of pointers originating with a root pointer to be "live."

Brief Summary Text (40):
Slow Pointer is a term specific to this invention disclosure which describes pointers that are implemented in such a way that they provide coordination with a background garbage collection task. Various implementations of slow pointers are possible. In general, fetching, storing, and indirecting through slow pointer variables is slower than performing the same operation on fast pointer variables.

Brief Summary Text (42):
Tending is a term of art describing the garbage collection process of examining a pointer to determine that the object it refers to is live and arranging for the referenced object to be subsequently scanned in order to tend all of the pointers contained therein.

Brief Summary Text (49):
1. Extensions to the standard Java byte code instruction set to enable efficient run-time isolation of pointer variables from non-pointer variables. The extended byte codes are described as the PERC instruction set.

Brief Summary Text (52):
of the PERC instruction set. The Java run-time stack is replaced by two stacks, one for non-pointer and the other for pointer data. Further, the data structures enable efficient interaction between native methods, Java methods represented by byte code, and Java methods translated by a JIT compiler to native machine language. Performance tradeoffs are biased to give favorable treatment to execution of JIT-translated methods.

Drawing Description Text (2):
FIG. 1 illustrates the organization of thread memory, with each thread comprised of a C stack, and pointer stack, and a non-pointer stack, and each stack represented by multiple stack segments

Drawing Description Text (3):
FIG. 2 illustrates the header information attached to each dynamically allocated memory object for purposes of performing garbage collection. These header fields consist of Scan-List, Indirect-Pointer, Activity-Pointer, Signature-Pointer, and optional Finalize-Link pointers.

Drawing Description Text (6):
FIG. 5 illustrates the appearance of the pointer and non-pointer stack activation

frames immediately before calling and immediately following entry into the body of a Java method. The stacks are assumed to grow downward. In preparation for the call, arguments are pushed onto the stack. Within the called method, the frame pointer (fp) is adjusted to point at the memory immediately above the first pushed argument and the stack pointer (sp) is adjusted to make room for local variables to be stored on the stack.

Drawing Description Text (7):
FIG. 6 illustrates the internal organization of the local-variable region of the stack activation frame. This region includes application-declared locals (as declared in byte-code attributes for Java methods and as specified in the parameterization of BuildFrames( ), temporary variables (as might be required to represent the old values of the frame and instruction pointers), a run-time stack (to allow execution of push and pop operations within the method), and space for arguments to be pushed to other methods to be called from this method.

Drawing Description Text (26):
FIG. 25 provides the C declaration of the structure used internal to the PERC implementation to represent a PERC stack of non-pointers.

Drawing Description Text (31):
FIG. 30 provides a C macro definition of the LongJmpo macro, which is a version of the standard C longjmp( ) function specialized for the PERC virtual machine execution environment. Note that this macro makes use of perclongjmp( ) whose implementation is not provided. perclongjmp( ) expects as parameters a representation of the machine's registers including its instruction pointer, the value of the pointer stack pointer, the value of the non-pointer stack pointer, and the return value to be returned to the point of the JIT version of setjmp( ).

Drawing Description Text (32):
FIG. 31 provides a C declaration of the structure used internal to the PERC implementation to represent a PERC stack of pointers.

Drawing Description Text (33):
FIG. 32 illustrates the signature structure used to represent the memory layout of heap-allocted objects. total.sub.-- length is the total number of words comprising the object, excluding the object's header words, but including its signature if the signature happens to be appended to the end of the data. All pointers are assumed to be word aligned within the structure. Use last.sub.-- descriptor to symbolically represent the word offset of the last word within the corresponding object that might contain a pointer. When the garbage collector scans the corresponding object in search of pointers, it looks no further than the word numbered last.sub.-- descriptor. type.sub.-- code comprises a 2-bit type tag in its most significant bits, with the remaining 30 bits representing the value of last.sub.-- descriptor. bitmap is an array of integers with each integer representing 32 words of the corresponding object, so there are a total of ceiling(last.sub.-- descriptor/32) entries in the array. (bitmap[0]&0.times.01), which represents the first word of the corresponding object, has value 1 if and only if the first word is a pointer.

Drawing Description Text (41):
FIG. 40 provides C macros for conversion between integer offsets and actual derived pointer values and for obtaining the actual address of the constant-pool object. These macros are used to improve the efficiency of access to instruction, stack, and constant-pool memory.

Drawing Description Text (45):
FIG. 44 provides C helper macros for use by application code to coordinate with the dispatcher. TendPointerStack( ), used by SaveThreadState( ), rescans the portion of the pointer stack that is bounded below by .sub.-- gc.sub.-- ps.sub.-- low.sub.-- water and above by .sub.-- psp.

Drawing Description Text (56):
FIG. 55 provides C macros for use in maintaining activation frames on the PERC pointer and non-pointer stacks. The StackOverflowCheck( ) macro is executed each time these stacks expand. The AdjustPSPAndZeroOutLocals( ) macro is executed to zero out the new pointers allocated on the PERC pointer stack. The AdjustLowWaterMacro( ) macro executes each time an activation frame is removed from the pointer stack. The low-water mark identifies the lower limit on the range of the pointer stack that has to be scanned when the task is preempted.

Drawing Description Text (65):
FIG. 64 provides the definitions of C macros for manipulation of the PERC pointer stack.

Drawing Description Text (66):
FIG. 65 provides the definitions of C macros for manipulation of the PERC non-pointer stack.

Drawing Description Text (95):
FIG. 94 illustrates the PERC non-pointer stack activation frame for JIT-generated code. Upon entry into the JIT function, the non-pointer stack pointer (npsp) points to the list of incoming arguments, and the return address is stored in the slot "above" the top-of-stack entry. The prologue of JIT-compiled method subtracts a JIT-computed constant from npsp to make room on the non-pointer stack for saved machine registers, local variables, and outgoing arguments.

Drawing Description Text (100):
FIG. 99 illustrates the implementation of the stackOverflow( ) help routine, which is invoked whenever the PERC pointer or non-pointer stacks are close to overflowing.

Detailed Description Text (12):
Methods represented as byte codes are interpreted by the PERC virtual machine. The interpreter, known throughout this invention disclosure as pvm() (for PERC virtual machine), uses three stacks for execution: (1) the traditional C stack, (2) an explicitly managed stack for representation of PERC pointer values, and (3) an explicitly managed stack for representation of PERC non-pointer values. The C stack holds C-declared local variables and run-time state information associated with compiler generated temporaries. The PERC pointer stack holds the pointer arguments passed as inputs to the method, pointer local variables, temporary pointers pushed during expression evaluation, and pointer values pushed as arguments to methods called by the current method. The PERC non-pointer stack holds non-pointer arguments passed as inputs to the method, non-pointer local variables, temporary non-pointer values pushed during expression evaluation, and non-pointer values pushed as arguments to be called by this method. The pointer and non-pointer stack activation frames are illustrated in FIG. 5 and FIG. 6.

Detailed Description Text (14):
Methods that have been translated to native machine code use only two stacks: the PERC pointer stack and the PERC non-pointer stack. The benefit of using only two rather than three stacks is that this reduces the overhead of stack maintenance associated with each method invocation. The activation frames for the two stacks are structured as illustrated in FIG. 94. However, the amount of information stored in the "temporaries" segment of the activation frame differs between JIT-compiled methods and byte-code methods.

Detailed Description Text (20):
The PERC implementation represents every PERC object with a data structure patterned after the templates provided in FIG. 15, FIG. 16, and FIG. 24. In all of these structures, the second field is a pointer to a MethodTable data structure

(see FIG. 23). The PERC execution environment maintains one MethodTable data structure for each defined object type. All instantiated objects of this type point to this shared single copy. The jit.sub.-- interfaces array field of the MethodTable structure has one entry for each virtual method supported by objects of this type. The mapping from method name and signature to index position is defined by the class loader, as described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley. To execute the JIT version of a PERC method using a virtual method lookup, branch to the code represented by jit.sub.-- interfaces[method.sub.-- index]. Normally, the JIT version of the byte code will only be invoked directly from within another JIT-compiled method. If a native or untranslated byte-code method desires to invoke another method using virtual method lookup, the search for the target method generally proceeds differently. First, we find the target object's MethodTable data structure (as above) and then follow the methods pointer to obtain an array of pointers to Method objects. Within the Method object, we consult the access.sub.-- flags field to determine if the target method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be implemented by byte codes. See FIG. 49, FIG. 45, and FIG. 46.

Detailed Description Text (25):
When the method to be invoked is declared as static within the corresponding object (meaning that the method operates on class information rather than manipulating variables associated with a particular instance of the corresponding class), the Java compiler treats this as an invokeStatic method. Execution of static methods is identical to execution of special methods except that there is no implicit pointer to "this" passed as an argument to the called method. See FIG. 47, FIG. 45, and FIG. 46.

Detailed Description Text (27):
When a method is invoked through an interface declaration, the called method's name and signature is stored as part of the calling method's code representation. The compiler ensures that the object to be operated on has a method of the specified name and signature. However, it is not possible to determine prior to run time the index position within the method table that holds the target method. Thus it is necessary to examine the target object's mtable field, which points to the corresponding MethodTable structure. We follow the MethodTable's methods pointer to find an array of pointers to Method structures. And we search this array for a method that matches the desired name and signature. Once found, we invoke this method. We examine the Method object that represents the target procedure and consult its access.sub.-- flags field to determine if the method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be represented as byte code. See FIG. 50, FIG. 51, FIG. 45, and FIG. 46.

Detailed Description Text (30):
Note that native methods and pvm(), which interprets byte-code methods, use the same stack organization. Thus, calling another method from a native method is the same as calling the method from within the pvm() interpreter. In both cases, the caller invokes the callee by passing appropriate parameters to one of several available invocation routines, all of which are written primarily in C. These invocation routines consult internal fields within the Method structure that describes the callee to determine whether the callee is implemented as byte codes, the JIT translation of byte codes, or a native method (See FIG. 46). The invocation routine adjusts the stack and other state information as necessary in order to transfer control to the called method. When the called method returns, the invocation routine restores the stack and other state information to once again support the execution mode of the calling method. To call a byte-code method, the invocation routine saves the offset of the old frame and stack pointers in local C variables, sets up the callee's activation frames (See FIG. 5), and calls pvm(), passing a pointer to the called method's Method structure as the only argument. To

call a native method, the invocation routine saves the offsets of the old stack and frame pointers, sets up the native method's activation frames (See FIG. 5), and calls (*Method.native)(). To call a JIT-translated method, the invocation routine sets up the callee's activation frames (See FIG. 5), pushes the current C frame pointer onto the C stack and then saves the current value of the C stack pointer in the c.sub.-- sp field of the currently executing thread's Thread data structure, copies the current values of the .sub.-- psp and .sub.-- npsp variables into machine registers dedicated to these purposes (effectively making the PERC stacks become the run-time execution stacks), and branches to (*Method.jit.sub.-- interface)(), leaving the return address in the stack slot above the top-of-stack entry on the non-pointer stack. See FIG. 94 for an illustration of the non-pointer stack activation frame as it is organized during execution of JIT code.

Detailed Description Text (35):
From within the implementation of the pvmo and within native methods, the standard protocol for invoking other methods depends on the type of the call. A virtual method invocation vectors to the corresponding code by way of the target object's method table. The object to which the method corresponds is passed implicitly on the run-time stack. To invoke a virtual method, first push a pointer to the target object onto the pointer stack and then push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeVirtual (), passing as arguments pointers to the String objects that represent the class name and the target method's name and signature (See FIG. 49):

Detailed Description Text (36):
void invokeVirtual(String *class.sub.-- name, String *method.sub.-- name.sub.-- and.sub.-- sig); Note that invokeVirtual() must do a string search within the class representation to find the selected method. This is potentially a costly operation and we would prefer to avoid this cost when possible. When byte code is first loaded into our system, we perform this lookup and save the result, represented by a pointer to a Method structure, within the constant pool. Implementers of native methods may design similar optimizations. There are two mechanisms available to implementers of native methods for the purposes of looking up Method objects: findMethod() and getMethodPtr(). Both of these functions return a pointer to the corresponding Method object. With findMethod(), the desired method is described by a pointer to the known Class object and a String pointer to the method's name and signature. With getMethodPtr(), the desired method is described by String representations of the class name and of the method's name and signature. Prototypes for both functions are provided below:

Detailed Description Text (40):
Within the Method structure, information is available which characterizes the number of pointer arguments of this particular method and the offset of this method within the object's method table (see FIG. 22). To invoke a virtual function without incurring the overhead of a string method lookup, use the FastInvokeVirtualo macro, prototyped below (See FIG. 45):

Detailed Description Text (48):
At the API level, invoking an interface is similar to invoking a virtual or non-virtual method. First push a pointer to the target object onto the pointer stack and then push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeinterface(), passing as arguments String objects representing the name of the class and the name and signature of the method within the class, as prototyped below (See FIG. 50):

Detailed Description Text (52):
Note in the above that the purpose of the template argument is to allow FastInvokeInterface to determine the name and signature of the method that it must search for in the object found num.sub.-- ptr.sub.-- args slots from the current top-of-stack pointer on the PERC pointer stack.

Detailed Description Text (54):
A static method is one that makes use only of information that is associated with the corresponding class (rather than instances of the class). When a static method is invoked, there is no "target object" pushed onto the stack. To call a static method, push all of the method's arguments onto the pointer and non-pointer stacks, depending on their types. Then call invokeStatic(), passing as arguments String objects representing the name of the class and the name and signature of the method within the class, as prototyped below (See FIG. 47):

Detailed Description Text (61):
2. Copies the register-held psp and npsp registers into global memory locations .sub.-- psp and .sub.-- npsp. Then assigns sp and fp (the machine's stack and frame pointer registers) to reflect the current C-stack context, as represented by .sub.-- current.sub.-- thread.fwdarw.c.sub.-- sp.

Detailed Description Text (62):
3. Copies the return address off the non-pointer stack (See FIG. 94) and saves its value in a slot within the C stack frame.

Detailed Description Text (63):
4. Calculates and assigns values to .sub.-- pfp (pointer stack frame pointer) and .sub.-- npfp (non-pointer stack frame pointer), based on the current values of the corresponding stack pointers and the number of arguments of each type. The stack activation frames are arranged as illustrated in FIG. 5. Additionally, we adjust the pointer and non-pointer stack pointers to make room for the local variables that are required to execute the method, as represented by the max.sub.-- ptr.sub.-- locals and max.sub.-- non.sub.-- ptr locals fields of the corresponding Method structure.

Detailed Description Text (65):
6. For coordination with the garbage collector, we keep track of how high the pointer stack has grown during the current execution time slice. Since the stack grows downward, the high-water mark is represented by the minimum value of .sub.-- psp.

Detailed Description Text (66):
7. Calls pvm(), passing as a C argument a pointer to the Method object that describes the segment of code to be executed.

Detailed Description Text (70):
11. For coordination with the garbage collector, we keep track of how low the pointer stack has shrunk during the current execution time slice. Since the stack grows downward, the low-water mark is represented by the maximum value of .sub.-- pfp.

Detailed Description Text (76):
To support real-time performance, garbage collection runs asynchronously, meaning that the garbage collection thread interleaves with application code in arbitrary order. To support accurate garbage collection, it is necessary for the garbage collector to always be able to distinguish a thread's pointer variables (including stack-allocated variables and variables held in machine registers) from the thread's non-pointer variables.

Detailed Description Text (77):
To require each thread to maintain all pointers in variables that are at all times easily identifiable by the garbage collector imposes too great an overhead on overall performance. Thus, the PERC virtual machine described in this invention disclosure implements the following compromises:

Detailed Description Text (79):
2. Between preemption points, the thread is allowed to hold pointers in variables that may not be visible to the garbage collector. In this disclosure, we characterize such variables as "fast pointers." Fast pointers are typically declared in C as local variables, and may be represented either by machine registers or slots on the C stack.

Detailed Description Text (80):
3. Pointer variables that are visible to the garbage collector are known throughout this disclosure as "slow pointers". Slow pointers are typically represented by locations on the PERC pointer stack and by certain C-declared global variables identified as "root pointers".

Detailed Description Text (81):
4. Immediately following each preemption, the thread must consider all of its fast pointers to be invalid. In preparation for each preemption, the thread must copy the values of essential fast-pointer variables into slow pointers. Following each preemption, essential fast pointers are restored from the values previously stored in slow pointer variables. Note that, while the thread was preempted, a drefragmenting garbage collector might have relocated particular objects, requiring certain pointer values to be modified to reflect the corresponding objects' new locations.

Detailed Description Text (82):
5. Each C function in the virtual machine implementation is identified as either preemptible or non-preemptible. Before calling a preemptible function, the caller must copy all of its essential fast pointers into slow pointers. When the called function returns, the caller must restore the values of these fast-pointer variables by copying from the slow-pointer variables in which their values were previously stored. Throughout this disclosure, we refer to preemptible functions as "slow functions" and to non-preemptible functions as "fast functions."

Detailed Description Text (86):
1. Do not coerce pointers to integers or integers to pointers.

Detailed Description Text (87):
2. Do not perform any pointer arithmetic unless specifically authorized to do so (e.g. special techniques have been enabled to support efficient instruction and stack pointer operations).

Detailed Description Text (88):
3. Do not store tag information (e.g. low-order bits of a word) within memory locations that are identified as pointers to the garbage collector.

Detailed Description Text (89):
4. Do not store pointers to the C static region or to arbitrary derived addresses in locations identified as garbage-collected pointers, except that pointers to objects residing in the ROMized region are allowed. Note: A derived address is a location contained within an object. The garbage collector assumes all pointers refer to the base or beginning address of the referenced object.

Detailed Description Text (91):
6. When declaring fields and variables that point to the C static region, identify such fields as non-pointers insofar as garbage collection is concerned.

Detailed Description Text (92):
7. Pointers to garbage-collected objects cannot be stored in the C static region unless such pointers have been registered as root pointers.

Detailed Description Text (96):

Given that (base) (field.sub.-- expr) is an expression representing a pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, and that field.sub.-- type represents the type of (base) (field.sub.-- expr), return the fast pointer that represents this heap pointer's value. In the process, we may have to "tend" the pointer's value. Optionally, we may overwrite in place the value of (base) (field.sub.-- expr), so this expression should be a C 1-value. Sample usage:

Detailed Description Text (99):
Given that (base) (field.sub.-- expr) is an expression representing a pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, that field.sub.-- type represents the type of (base) (field.sub.-- expr), and that field.sub.-- value is also of type field.sub.-- type, assign field.sub.-- value to (base) (field.sub.-- expr). In the process, we may have to "tend" field.sub.-- value. Note that (base) (field.sub.-- expr) must be a C 1-value. Sample usage: SetHeapPtr(.sub.-- current.sub.-- thread, .fwdarw.current.sub.-- exception, Thread *, Object *, new.sub.-- exception);

Detailed Description Text (101):
Given that (base) (field.sub.-- expr) is an expression representing a non-pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, and that field.sub.-- type represents the type of (base) (field.sub.-- expr), return the non-pointer value that represents this heap location's value. Sample usage:

Detailed Description Text (104):
Given that (base) (field.sub.-- expr) is an expression representing a non-pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, that field.sub.-- type represents the type of (base) (field.sub.-- expr), and that field.sub.-- value is also of type field.sub.-- type, assign field.sub.-- value to (base) (field.sub.-- expr). Note that (base) (field.sub.-- expr) must be a C 1-value. Sample usage:

Detailed Description Text (114):
With certain garbage collection techniques, it is possible that two fast pointer objects refer to the same object even though their pointer values are different. This might occur, for example, if an object is being copied in order to compact live memory and one pointer refers to the original location of the object and the other pointer refers to the new copy of the object. Programmers should use the SameObject() macro to compare fast pointers for equality. This macro returns non-zero if and only if its two pointer arguments refer to the same object.

Detailed Description Text (142):
5.3 Manipulation of the Pointer Stack

Detailed Description Text (143):
Native methods and other C functions run fastest if they avoid frequent copying of values between local variables (stored on the PERC pointer stacks) and C-declared fast pointers. But pointers stored in C-declared variables are not necessarily preserved across preemption of the thread. Thus, it is necessary for the application code to copy from C-declared variables to macro-declared variables before each preemption. The following macros are used to manipulate the pointer stack.

Detailed Description Text (145):
Push the fast pointer fptr onto the pointer stack.

Detailed Description Text (147):
Pop the fast pointer off of the pointer stack.

Detailed Description Text (149):
Return the fast pointer that is currently offset slots away from the top of the stack.

Detailed Description Text (150):
PeekPtr(0) is the top pointer-stack element.

Detailed Description Text (154):
Return as a fast pointer the local pointer at the specified offset. The first pointer argument is at offset 0. The second pointer argument is at offset 1, and so on.

Detailed Description Text (156):
Set the local pointer at the specified offset to the value of fastptr. Offsets are specified as described in the GetLocalFastPtr() macro.

Detailed Description Text (158):
Adjust the pointer stack pointer (psp) by offset entries. If offset is positive, space representing offset pointers is removed from the stack. ShrinkPS should not be used with a negative offset as this might create a situation in which pointers have garbage values.

Detailed Description Text (159):
5.4 Manipulation of the Non-Pointer Stack

Detailed Description Text (160):
The non-pointer stack holds integers, 8-byte long integers, floating point values, and 8-byte double-precision floating point values. The following macros are suggested for manipulation of the non-pointer stack.

Detailed Description Text (162):
Push integer val onto the non-pointer stack.

Detailed Description Text (164):
Push floating point value val onto the non-pointer stack.

Detailed Description Text (166):
Push 8-byte long value val onto the non-pointer stack.

Detailed Description Text (168):
Push 8-byte double precision value val onto the non-pointer stack.

Detailed Description Text (170):
Pop a single integer from the top of non-pointer stack.

Detailed Description Text (172):
Pop a single floating point value from the top of the non-pointer stack.

Detailed Description Text (174):
Pop a single 8-byte long value from the top of the non-pointer stack.

Detailed Description Text (176):
Pop a single 8-byte double precision value from the top of the non-pointer stack.

Detailed Description Text (178):
Given that off words have been pushed onto the non-pointer stack on top of integer item n, return n.

Detailed Description Text (180):
Given that off words have been pushed onto the non-pointer stack on top of floating

point value x, return x.

Detailed Description Text (182):
Given that off words have been pushed onto the non-pointer stack on top of 8-byte long value m, return m.

Detailed Description Text (184):
Given that off words have been pushed onto the non-pointer stack on top of 8-byte double-precision value y, return y.

Detailed Description Text (186):
Given that off words have been pushed onto the non-pointer stack on top of the integer slot representing n, overwrite this slot with val.

Detailed Description Text (188):
Given that off words have been pushed onto the non-pointer stack on top of the floating point slot representing x, overwrite this slot with val.

Detailed Description Text (190):
Given that off words have been pushed onto the non-pointer stack on top of the 8-byte long integer slot representing m, overwrite this slot with val.

Detailed Description Text (192):
Given that off words have been pushed onto the non-pointer stack on top of the 8-byte double precision slot representing y, overwrite this slot with val.

Detailed Description Text (194):
Given that off words precede the integer variable j within the local non-pointer stack activation frame, return the value of j.

Detailed Description Text (196):
Given that off words precede the floating point variable f within the local non-pointer stack activation frame, return the value of f.

Detailed Description Text (198):
Given that off words precede the 8-byte long integer variable 1 within the local non-pointer pointer stack activation frame, return the value of 1.

Detailed Description Text (200):
Given that off words precede the double precision floating point variable x within the local non-pointer stack activation frame, return the value of

Detailed Description Text (203):
Given that off words precede the integer variable j within the local non-pointer stack activation frame, set the value of j to val.

Detailed Description Text (205):
Given that off words precede the floating point variable f within the local non-pointer stack activation frame, set the value of f to val.

Detailed Description Text (207):
Given that off words precede the 8-byte long integer variable 1 within the local non-pointer stack activation frame, set the value of 1 to val.

Detailed Description Text (209):
Given that off words precede the double precision floating point variable x within the local non-pointer stack activation frame, set the value of x to val.

Detailed Description Text (211):
Adjust the non-pointer stack pointer (npsp) by offset entries. If offset is

positive, space representing offset pointers is removed from the stack. If offset is negative, the specified number of stack slots are added to the stack.

Detailed Description Text (213):
Strict partitioning between fast and slow pointers, and requiring all heap memory access to be directed by way of heap access macros imposes a high overhead. Certain data structures are accessed so frequently that the PERC implementation treats them as special cases in order to improve system performance. In particular, the following exceptions are supported:

Detailed Description Text (214):
1. Note that the PERC stacks dedicated to representation of pointer and non-pointer data respectively are heap allocated. According to the protocols described above, every access to PERC stack data should be directed by way of heap access macros. Since stack operations are so frequent, we allow direct access to stack data using traditional C pointer indirection. This depends on the following:

Detailed Description Text (215):
a. The stack pointers are represented by C global variables declared as pointers. Access to stack data uses C pointer indirection, without enforcement of special read or write barriers. (See FIG. 64 and FIG. 65)

Detailed Description Text (216):
b. Each time the task is preempted, the global variables representing the currently executing thread's stack pointers are saved by the run-time dispatcher in the thread's structure representation (See FIG. 53 and FIG. 44). Note the use of the GetSPOffset() macro (See FIG. 40).

Detailed Description Text (217):
c. Each time a task is scheduled for execution, the dispatcher sets the global stack pointer variables to represent the newly dispatched thread's stack pointers (See FIG. 53 and FIG. 44). Note the use of the GetSP() macro (See FIG. 40).

Detailed Description Text (218):
d. During execution of a time slice, the thread's pointer stack is assumed to hold fast pointers. However, when the thread is preempted, the garbage collector needs to see the stack object's contents as slow pointers. When the thread is preempted, the dispatcher scans that portion of the stack that has been modified during the current time slice in order to convert all fast pointers to slow pointers (See FIG. 53 and FIG. 44). We maintain a low-water mark representing a bound on the range of stack memory that has been impacted by execution of the task during its current time slice to reduce the need for redundant stack scanning.

Detailed Description Text (220):
a. The instruction pointer is represented by a fast pointer declared within the implementation of pvm(). Upon entry into pvm(), this variable is initialized using the GetPC() macro, which expects as arguments a pointer to the ByteString object that represents the method's code and the instruction offset within this method's code (See FIG. 40).

Detailed Description Text (222):
c. After being preempted (or after returning from a function that may have been preempted), the instruction pointer is recomputed by using GetPC().

Detailed Description Text (223):
3. During interpretation of byte-code methods, the constant pool is frequently accessed. Rather than incurring the overhead of a standard heap access macro, we obtain a trustworthy C pointer to the constant pool data structure and refer directly to its contents. For this purpose, we use the GetCP() macro (See FIG. 40). C subscripting expressions based on the value returned by GetCP() are considered

valid up to the time at which the thread is next preempted. Following each preemption, the pointer must be recomputed through another application of the GetCP () macro.

Detailed Description Text (235):
2. Native methods: Like the virtual machine, each invocation of a native method must be preceded by the preparation of PERC stack activation frames. The format of the activation frames and the protocol for allocation of local pointers is exactly the same for native methods as for pvm().

Detailed Description Text (238):
a. Before calling a slow procedure, all fast pointers that are considered to be live must be saved on the PERC pointer stack.

Detailed Description Text (239):
b. Pointer arguments can be passed either on the PERC pointer stack or on the C stack (as regular arguments). Any live arguments passed on the C stack must be saved on the PERC pointer stack prior to calling another slow procedure or reaching a voluntary preemption point.

Detailed Description Text (243):
AllocLocalPointers(num.sub.-- ptrs, first.sub.-- local.sub.-- offset, ptr.sub.-- stack.sub.-- growth, non.sub.-- ptr.sub.-- stack.sub.-- growth); num.sub.-- ptrs specifies the number of pointers for which space is to be reserved within the local activation frame. first.sub.-- local.sub.-- offset is an integer variable that is initialized by this macro to represent the location of the first local variable relative to the beginning of this function's activation frame. ptr.sub.-- stack.sub.-- growth and non.sub.-- ptr.sub.-- stack.sub.-- growth represent the maximum additional stack expansion that might take place during execution of the corresponding stack (through push operations and/or the allocation of stack slots for outgoing arguments). The information provided by these last two arguments is used to perform stack overflow checking and to adjust the pointer stack high-water mark.

Detailed Description Text (244):
To find the offset of the top-of-stack entry within an activation frame that has no local pointers, use the following:

Detailed Description Text (249):
num.sub.-- ptr.sub.-- args and num.sub.-- ptr.sub.-- locals represent the number of incoming pointer arguments and the number of local pointer variables respectively. These variables determine the amount by which the pfp and psp pointers must be adjusted in order to establish the pointer stack activation frame. ptr.sub.-- stack.sub.-- growth is the number of additional stack slots (beyond the slots set aside for locals and arguments) required on the pointer stack to support execution of this slow procedure. This variable is used to check for pointer stack overflow, if such a check is desired. num.sub.-- non.sub.-- ptr.sub.-- args, num.sub.-- non.sub.-- ptr.sub.-- locals, and non.sub.-- ptr.sub.-- stack.sub.-- growth serve the same roles with respect to the non-pointer stack as the corresponding pointer stack variables.

Detailed Description Text (253):
DestroyFrames(num.sub.-- pointers, num.sub.-- non.sub.-- pointers)

Detailed Description Text (254):
DestroyFrames() removes all but num pointers words from the pointer stack and all but num.sub.-- non.sub.-- pointers words from the non-pointer stack. Note that a DestroyFrames() invocation must occur on each control-flow path that reaches either the end of the function's body or a return statement. Prior to invocation of the DestroyFrames() macro, the application code should store the return value into the

0.sup.th slot of the corresponding stack frame.

Detailed Description Text (256):
PrepareJavaFrames(). In preparation for calling the pvm(), as is done within the invoke routines (invokeVirtual(), invokeSpecial(), invokeStatic(), and invokeInterface()) and within byte-code stubs, it is necessary to construct the activation frames for the PERC pointer and non-pointer stacks. This is done by executing the PrepareJavaFrames() macro, with parameters similar to what was described above for

Detailed Description Text (262):
PrepareNativeFrames(). In preparation for calling a native method, as is done within the invoke routines (invokeVirtual(), invokeSpecial(), invokeStatic(), and invokeInterface()) and within byte-code stubs, it is necessary to construct the activation frames for the PERC pointer and non-pointer stacks. This is done by executing the PrepareNativeFrames() macro, with parameters similar to what was described above for BuildFrames():

Detailed Description Text (268):
ReclaimFrames(num.sub.-- pointers, num.sub.-- non.sub.-- pointers);

Detailed Description Text (271):
AdjustLowWaterMark. Both ReclaimFrames() and DestroyFrames() make use of the AdjustLowWaterMark() macro, which is defined in FIG. 55. The purpose of this macro is to keep track of the lowest point to which the pointer stack has shrunk during execution of the current time slice. When this task is preempted, all of the pointers between the low-water mark and the current top-of-stack pointer are tended. By tending these pointers at preemption time, it is not necessary to enforce the normal write barrier with each update to the pointer stack.

Detailed Description Text (273):
The PERC Virtual Machine describes the C function that interprets Java byte codes. This C function, illustrated in FIG. 68, is named pvm(). The single argument to pvm () is a pointer to a Method structure, which includes a pointer to the byte-code that represents the method's functionality. Each invocation of pvm() executes only a single method. To call another byte-code method, pvm() recursively calls itself. Note that pvm() is reentrant. When multiple Java threads are executing, each thread executes byte-code methods by invoking pvm() on the thread's run-time stack.

Detailed Description Text (274):
The implementation of pvm() allocates space on the PERC pointer stack for three pointer variables. These pointers, known by the symbolic names pMETHOD, pBYTECODF, and pCONSTANTS, represent pointers to the method's Method structure, the StringOfBytes object representing its byte code, and the constant-pool object representing the method's constant table respectively. During normal execution of pvm(), the values of these variables are stored in the C locals method, bytecode, and cp respectively. Before preemption, and before calling preemptible functions, pvm() copies the contents of these C variables onto the PERC pointer stack. In preparation for executing the byte codes representing a byte-code method, pvm() checks to determine if the method has any exception handlers. If the method is synchronized, the lock will have been obtained by the fastInvokeo routine prior to calling pvm() (see FIG. 46). However, fastInvoke() does not set an exception handler to release the lock if the code is aborted by the raising of an exception. For this reason, pvm() sets an exception handler if the method is synchronized, so that it can release the lock before rethrowing the exception to the surrounding context.

Detailed Description Text (283):
The IADD instruction removes the top two elements from the non-pointer stack, both of which are known to represent integers, adds these two integer values, and stores

the sum onto the top of the same stack. This is illustrated in FIG. 69.

Detailed Description Text (284):
The AASTORE instruction removes from the pointer stack a reference to an array and a reference to an object to be inserted into an array, and removes from the non-pointer stack an integer index representing the position within the array that is to be overwritten with the new value. This instruction makes sure that the array subscript is within bounds and makes sure that the value to be inserted into the array is of the proper type. Then it stores the reference into the array at the specified index position, as illustrated in FIG. 70.

Detailed Description Text (285):
The FCMPL instruction removes the top two elements from the non-pointer stack, both of which are known to represent floating point numbers, compares these two values, and stores an integer representing the result of comparison onto the same stack. The result is encoded as 0 if the two numbers are equal, -1 if the first is less than the second, and 1 if the first is greater than the second. The implementation of FCMPL, is illustrated in FIG. 71.

Detailed Description Text (286):
The IFEQ instruction (See FIG. 72) branches to the byte-code instruction obtained by adding the two-byte signed quantity which is part of the instruction encoding to the current value of the program pointer if the top of the non-pointer stack, which is known to represent an integer, holds the value 0. Note that the PVMPreemptionPointo macro appears before the break statement. pvm() allows itself to be preempted at this point. In general, pvm() considers each byte-code instruction which may cause control branching to be a preemption point. This guarantees that there is at least one preemption point in each byte-code loop.

Detailed Description Text (287):
The JSR instruction (See FIG. 73) jumps to a subroutine by branching to the byte-code instruction obtained by adding the two-byte signed quantity which is part of the instruction encoding to the current value of the program counter and pushing the return address onto the non-pointer stack. Note that the return address is represented as the integer offset within the current method's byte code rather than an actual pointer. This is because the garbage collector does not deal well with pointers that refer to internal addresses within objects rather than to the objects' starting addresses. Note also that the JSR instruction also invokes the PVMPremptionPoint() macro.

Detailed Description Text (289):
The TABLESWITCH instruction (See FIG. 75) is used to efficiently implement switch statements in which most of the various cases are represented by consecutive integers. The immediate-mode operands of this instruction are encoded as (1) padding to align the next operand at an address that is a multiple of 4 bytes, (2) a low integer value representing the first integer in the range of cases, (3) a high integer value representing the last integer in the range of cases, (4) the integer representing the byte-code offset of the code that represents the default case, and (5) (high+1-low) integers representing the byte-code offsets of the code that implements each of the cases. This instruction removes the top entry, which is known to be an integer, from the non-pointer stack and uses this value to index into the branch table in order to compute the address of the next instruction to be executed. Note that TABLESWITCH invokes the PVMPreemptionPoint() macro.

Detailed Description Text (290):
The LOOKUPSWITCH instruction (See FIG. 76) is used to implement switch statements in which the cases are not consecutive integers. The immediate-mode operands of this instruction are encoded as (1) padding to align the next operand on an address that is a multiple of 4 bytes, (2) an integer representing the total number of cases, (3) the integer representing the byte-code offset of the code that

represents the default case, and (4) pairs of key values combined with instruction offsets for each of the cases identified in field number 2. This instruction removes the top entry from the non-pointer stack, which is known to be an integer, and searches for this value among the cases represented in its encoding. Note that LOOKUPSWITCH invokes the PVMPreemptionPoint() macro.

Detailed Description Text (291):
The IRETURN instruction (See FIG. 77) is used to return an integer from the currently executing method. This instruction pops the integer value to be returned from the top of the non-pointer stack and stores the integer value into the 0th slot of the non-pointer stack's current activation frame. Then it breaks out of the interpreter loop by using a goto statement.

Detailed Description Text (292):
The GETSTATIC.sub.-- QNP8 instruction (See FIG. 78) gets an 8-bit non-pointer value from the static area of the class corresponding to the field that is stored in the constant-pool table at the offset specified by this instruction's one-byte immediate-mode operand. The value fetched from the static field is pushed onto the non-pointer stack.

Detailed Description Text (293):
The PUTFIELLD.sub.-- Q instruction stores a value (provided on one of the PERC stacks) into the specified field of a particular object. A pointer to the object that contains the field is passed on the pointer stack. The two-byte immediate operand of this instruction indexes into the constant pool to find a 4-byte integer value. This integer value encodes the offset of the field within the object as the least significant 29 bits, an encoding of the number of bits to be updated if the field is not a pointer in the next two most significant bits, and a flag distinguishing pointer fields in the most significant bit. The implementation of this instruction is illustrated in FIG. 79.

Detailed Description Text (294):
The INVOKEVIRTUAL.sub.-- FQ instruction (See FIG. 80) invokes a virtual function. The method-table index is encoded as the first immediate-mode byte operand and the number of pointer arguments is encoded as the second immediate-mode byte operand. Note that most of the work associated with invoking the virtual method is performed by the FastInvokeVirtual() macro, which is illustrated in FIG. 45. Note also that pvm() saves and restores its state surrounding the method invocation.

Detailed Description Text (296):
Invocation of interfaces is performed by the INVOKEINTERFACE.sub.-- Q instruction (See FIG. 83). Invoking interfaces is inherently more complicated than the other forms of invocation because the method table of the target object must be searched for a method with a matching name and signature. It is not generally possible to map the name and signature to an integer index prior to execution of the instruction. The immediate-mode operands to this instruction are (1) a one-byte index into the constant pool table to obtain a pointer to a method that has the desired name and signature, (2) a one-byte operand representing the number of pointer arguments passed to the interface method, and (3) a one-byte guess as to the offset within the target object's method table at which the target method will be found. See the definition of the FastInvokeInterface() macro in FIG. 45.

Detailed Description Text (297):
The NEW.sub.-- Q instruction (See FIG. 84) allocates a new object. This instruction takes a two-byte immediate-mode operand, which is an index into the constant pool. The corresponding entry within the constant pool is a pointer to the Class object (See FIG. 16) that describes the type of the object to be allocated. The newly allocated object is pushed onto the pointer stack.

Detailed Description Text (298):

The NEWARRAY instruction (See FIG. 85) allocates a new array of non-pointer data. The type of the non-pointer data is encoded as a one-byte immediate-mode operand to the instruction. The size of the array is passed as an integer on the non-pointer stack. The newly allocated array is pushed onto the pointer stack.

Detailed Description Text (299):
The ANEWARRAY.sub.-- Q instruction (See FIG. 86) allocates a new array of pointers. The type of the array entry is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the corresponding Class object. The size of the array is passed as an integer on the non-pointer stack. The newly allocated array is pushed onto the pointer stack.

Detailed Description Text (300):
The ATHROW instruction (See FIG. 87) throws the exception object that is on the top of the pointer stack. Note that this instruction causes control to longjmp out of the current pvm() activation. Where the exception is caught, the stacks will be truncated to the appropriate heights. Thus, it is not necessary to pop the thrown exception.

Detailed Description Text (301):
The CHECKCAST.sub.-- Q instruction (See FIG. 88) ensures that the top pointer stack element is of the appropriate type, where appropriate type is defined to mean that the type of the stack element is derived from the "desired" type. If it is not, this instruction throws an exception. The desired type is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the Class object that represents the desired type. Note that the NULL value is considered to match any reference type. If the top pointer stack value is of the appropriate type, the value is left on top of the pointer stack.

Detailed Description Text (302):
The INSTANCEOF.sub.-- Q instruction (See FIG. 89) removes the top pointer stack element and checks to see if it is of the appropriate type, where appropriate type is defined to mean that the type of the stack element is the "desired" type. If it is, this instruction pushes a 1 onto the non-pointer stack. If it isn't, this instruction pushes a 0 onto the non-pointer stack. The NULL value is considered to be of the appropriate type. The desired type is encoded as a two-byte immediate-mode operand which indexes into the constant-pool table to obtain a pointer to the Class object that represents the desired type.

Detailed Description Text (303):
The MONITORENTER instruction (See FIG. 90) removes the object reference on the top of the pointer stack and arranges to apply a semaphore-like lock on that object. If the object is already locked by another thread, the current thread is put to sleep until the object becomes unlocked. Note that the pvm()'s state is saved and restored surrounding the call to the enterMonitor() function, because that call may result in preemption of this thread. Note that if the entry on the top of the pointer stack is NULL, this instruction throws an exception. The MONITOREXIT instruction (See FIG. 91) removes the object reference on the top of the pointer stack and arranges to remove its semaphore-like lock on that object. If the object has been locked multiple times by this thread, this instruction simply decrements the count on how many times this object has been locked rather than removing the lock. As with the MONITORENTER instruction, pvm( )'s state is saved and restored surrounding the call to the exitMonitor() function and this instruction throws an exception of the top of the pointer stack is NULL.

Detailed Description Text (307):
1. The current values of the pointer stack and frame pointers.

Detailed Description Text (308):
2. The current values of the non-pointer stack and frame pointers.

Detailed Description Text (309):
3. The explicitly saved value of the C stack pointer, for situations in which the exception handling context is established from within JIT-compiled code. While JIT-compiled code is executing, the C stack is not used, and the value of the C stack pointer is held in a special field of the corresponding Thread object.

Detailed Description Text (310):
4. A pointer to the surrounding exception handling context. The special supplemental information fields are stored within the PERCEnvironment data structure, which is illustrated in FIG. 26.

Detailed Description Text (314):
In the PERC implementation, every object has a HashLock pointer field, which is initialized to NULL. When either a lock or a hash value is needed for the object, a HashLock object (see FIG. 20) is allocated and initialized, and the HashLock pointer field is made to refer to this HashLock object. Note that each HashLock object has the following fields:

Detailed Description Text (316):
2. The u field is a union which can represent either a pointer to another HashLock object (in case this HashLock object is currently residing on a free list), or a pointer to the thread that owns this semaphore if the lock is currently set, or NULL if this object is not currently on a free list and the lock is not currently set.

Detailed Description Text (319):
Obtaining a hash value. When application code desires to obtain the hash value of a particular object, it invokes the native hashCode() method. This method consults the object's lock field. If this field is NULL, this method allocates a HashLock object, initializes its hash.sub.-- value field to the next available hash value, and initializes the object's lock pointer to refer to the newly alocated HashLock object. Then it returns the contents of the hash.sub.-- value field. If the lock field is non-NULL, hashCode() consults the hash.sub.-- value field of the corresponding HashLock object to determine whether a hash value has already been assigned. If this field has value 0, hashCode() overwrites the field with the next available hash value. Otherwise, the hash value has already been assigned. In all cases, the last step of hashCode() is to return the value of the hash.sub.-- value field.

Detailed Description Text (332):
Each thread is represented by a Thread object which includes instance variables representing the critical state information associated with the thread (See FIG. 36). One of the instance variables points to the jump buffer (PERCEnvironment) of the currently active exception handler context. Each thread maintains three stacks, one to represent C activation frames, one to represent non-pointer PERC arguments and local variables, and a third to represent PERC pointer arguments and local variables. JIT-generated code uses only the two PERC stacks. While executing JIT-generated code, the C stack pointer is stored in a Thread field so that the machine's stack-pointer register can refer to the non-pointer PERC stack. We desire to allocate small stacks so as to conserve memory. This is especially important for applications comprised of large numbers of threads. For reliability, we provide stack overflow checking and, in some cases, the capability of expanding stacks on the fly as necessary.

Detailed Description Text (333):
We say that the C stack segments contain no pointers, but this is not entirely true. Since the C activation frame contains return addresses, the stack contains pointers to code. And since the activation frame includes saved registers, it probably contains the saved values of frame and stack pointers. To avoid the

complications and efficiency hits that would be associated with the handling of these pointers by a relocating garbage collector, we require stack segments to be non-moving, except for one exception which is discussed below.

Detailed Description Text (334):
The C stack may also contain pointers to heap objects which were saved in registers or local variables within particular activation frames. The usage protocol requires that such variables be treated as dead insofar as the garbage collector is concerned.

Detailed Description Text (335):
All three kinds of stacks are represented by multiple stack segments. In general, each run-time stack is allowed to expand on the fly as necessary. Expansion occurs whenever a stack overflow is detected. Expansion consists of allocating a new stack segment, copying that portion of the original stack segment that is necessary to establish an execution context on the new stack segment (the incoming parameters, for example), adjusting links to represent the addition of the new stack segment and setting the corresponding stack pointer(s) to their new values. The data structures are illustrated in FIG. 1.

Detailed Description Text (337):
The sample implementation characterized by this invention disclosure uses operating system provided stack overflow checking and stack expansion for the C stack, and uses explicit software overflow checks for the PERC pointer and non-pointer stacks. The C stack overflow checking is performed using memory management hardware.

Detailed Description Text (348):
1. The watchdog task is written entirely in C. Thus, it does not make use of the PERC stack and frame pointer variables.

Detailed Description Text (354):
1. Registers as root pointers .sub.-- gc.sub.-- thread and .sub.-- dispatcher.sub.-- thread. These static variables identify the Thread objects that govern the garbage collection thread and the real-time dispatcher thread respectively.

Detailed Description Text (385):
11. Sets the global .sub.-- current.sub.-- thread pointer to refer to the dispatcher task.

Detailed Description Text (389):
When a new thread is created, the system allocates a C stack, a PERC non-pointer stack, and a PFRC pointer stack. The size of the C stack is determined as a run-time option (specified on the command line if the virtual machine is running in a traditional desktop computing environment). The size of the PERC pointer and non-pointer stacks is specified by compile-time macro definitions, defined to equal 1024 words per stack.

Detailed Description Text (392):
A compile-time constant represents a threshold test for proximity to the PERC stack overflow conditions (See P.sub.-- SAFETY.sub.-- PADDING and NP.sub.-- SAFETY.sub.-- PADDING in FIG. 55). Within the BuildFrames(), PrepareJavaFrames(), and AllocLocalPointers() macros, we test whether the current stack pointer is within this threshold of overflowing the corresponding stack. If so, we immediately create a new stack frame for execution of this procedure by:

Detailed Description Text (395):
3. Recursively calling this same procedure by way of a trampoline function which is responsible for restoring the stack to "normal" upon return from the recursive invocation. Note that certain code sequences may result in "thrashing" of the stacks in the sense that every time we call a particular procedure, we grow the

stack and every time the procedure returns, we shrink it. If we end up in a loop that repeatedly calls this procedure, we will find ourselves spending too much time managing the stack growth. A solution to this problem is to keep track of how frequently particular stacks need to be expanded. If a particular stack is expanded too frequently, then we will consider it worth our while to expand the stack contiguously. Contiguous expansion of the stack consists of creating a single larger stack segment that is large enough to represent multiple C stack segments and copying the first C stack segment onto this new stack. While copying the stack data, it is necessary to adjust stack pointers that refer to the stack. Primarily, this consists of the frame pointer information that might be stored on the C stack.

Detailed Description Text (397):
So-called fast pointers refer directly to the corresponding memory objects using traditional C syntax. Fast pointers are stored on the traditional C stack or in machine registers. They are not scanned by the garbage collector. Thus, it is very important to make sure that garbage collection occurs at times that are coordinated with execution of application threads. (If the garbage collector were to relocate an object "while" an application thread was accessing that object by way of a fast pointer, the application thread would become confused.) Each application thread is responsible for periodically checking whether the system desires to preempt it. The following macro serves this purpose:

Detailed Description Text (405):
Note that each time we preempt a task, we must be prepared to save and restore all of the fast pointers that are currently in use. However, in cases where a particular pointer variable is known to have been saved to the stack already, and has not been modified since it was last saved to the stack, it is possible to omit the save operation. The purpose of copying fast pointers into "local" pointer variable slots is to make them visible to the garbage collector. After the task has been preempted, the application task's fast pointers may no longer be valid. Thus, the application task must restore the fast-pointer variables by copying their updated values from the local pointer variables.

Detailed Description Text (408):
2. Rescanning all of the PERC stack pointer's data found between the stack's low-water mark and the current stack pointer. Then set the low-water stack mark to refer to the current stack activation frame. (The last of these two operations may be postponed until after this thread is resumed.)

Detailed Description Text (417):
In the first case, the protocol described immediately above ensures that local variables are in a consistent state at the moment the task is preempted. To handle the second case, we require that any C code in the run-time system that calls a non-fast function consider all of its fast pointers to have been invalidated by invocation of the non-fast function. Further, we require that the invocation of blocking system calls be surrounded by the PrepareBlockCall() and ResumeAfterBlockCall() macros, as shown below:

Detailed Description Text (431):
Native libraries are implemented according to a protocol that allows references to dynamic objects to be automatically updated whenever the dynamic object is relocated by the garbage collector. However, if these native libraries call system routines which do not follow the native-library protocols, then the system routines are likely to become confused when the corresponding objects are moved. To avoid this problem, programmers who need to pass heap pointers to system libraries must make a stable copy of the heap object and pass a pointer to the stable copy. The stable copy should be allocated on the C stack, as a local variable. If necessary, upon return from the system library, the contents of the stable copy should be copied back into the heap. Note that on uniprocessor systems a non-portable

performance optimization to this strategy is possible when invoking system libraries that are known not to block if thread preemption is under PERC's control. In particular, we can pass the system library a pointer to the dynamic object and be assured that the dynamic object will not be relocated (since the garbage collector will not be allowed to run) during execution of the system library routine.

Detailed Description Text (439):
8.3 Global Pointer Variables (Roots)

Detailed Description Text (440):
All global root pointers must be registered so that they can be identified by the garbage collector at the start of each garbage collection pass. These root pointers are independently registered using the RegisterRooto macro, prototyped below. Each root pointer must be registered before its first use.

Detailed Description Text (451):
Each heap-allocated object must be identified so that the garbage collector can determine which of its fields contain pointers. The standard technique for identifying pointers within heap objects is to provide a signature for each object. The signature pointer occupies a particular word of each object's header (See FIG. 2).

Detailed Description Text (452):
The signature structure is illustrated in FIG. 32. The total.sub.-- length field counts the total number of words in the corresponding object. The type.sub.-- code field comprises two kinds of information: a four-bit code identifying the kind of object and a twenty-eight-bit integer that identifies the word offset of the last pointer contained within this object. If there are no pointers contained within the object, the word offset has value zero. The most significant bit of type.sub.-- code is set to indicate that the corresponding object needs to be finalized. The next three most-significant bits encode the kind of object, as represented by the preprocessor constants in FIG. 33. These special constants are manipulated using the macros provided in FIG. 34.

Detailed Description Text (453):
Within the signature structure, bitmap is an array of bits with one bit representing each word of the corresponding object. The bit has value zero if the corresponding word is a non-pointer, and value one if the corresponding word is a pointer. The first word of the object is represented by (bitmap[0]& 0.times.01). The second word is represented by (bitmap[0]& 0.times.02). The thirty-third word is represented by (bitmap[1]& 0.times.01), and so forth. Bits are provided only up to the word offset of the last pointer. Note that multiple heap-allocated objects may share the same statically allocated signature structure.

Detailed Description Text (473):
The special preprocessor converts the signature macro to the following declaration: static int.sub.-- sig1234[ ]={5, Record .vertline.5, 0.times.01f,}; static struct Signature *.sub.-- sigClassFile=(struct Signature *).sub.-- sig1234; The codes used to identify fields within a structure are the same as the primitive C types: char, short, int, long, float, double. Note that we need not distinguish unsigned values. The ptr keyword represents pointers (the garbage collector does not need to know the type of the object pointed to).

Detailed Description Text (495):
Every PERC object begins with two special fields representing the object's lock and method tables respectively. See FIG. 23 for the declaration of MethodTable. The method table's first field is a pointer to the corresponding Class object. The second field is a pointer to an array of pointers to Method objects. The third field is a pointer to the JIT-code implementation of the first method, followed by

a pointer to the JIT-code implementation of the second method, and so on. The pointers to JIT-code implementations may actually be pointers only to stub procedures that interface JIT code to byte-code or native-code methods.

Detailed Description Text (496):
Allocation routines. When allocating memory from within a native method, the programmer provides to the allocation routine the address of a signature rather than simply the size of the object to be allocated. The Signature pointer passed to each allocate routine must point to a statically allocated Signature structure. The implementation of the PERC virtual machine allocates a static signature for each class loaded. Once this static signature has been created, all subsequent instantiations of this class share access to this signature.

Detailed Description Text (499):
Note that every allocated object is tagged according to which real-time activity allocated it. This is necessary in order to allow the run-time system to enforce memory allocation budgets for each activity. Allocations performed by traditional Java applications that are not executing as part of a real-time activity are identified by a null-valued Activity pointer. All of the allocate routines consult the Thread referenced by.sub.-- current.sub.-- thread to determine which Activity the current thread belongs to.

Detailed Description Text (500):
In some cases, such as when a dynamically allocated object contains union fields that contain pointers only some of the time, it is necessary to allocate a private copy of the signature along with the actual object. To minimize allocation overhead, both the signature and the data are allocated as a single contiguous region of memory using the following allocation routine, which assumes that its sp argument points to static memory:

Detailed Description Text (506):
In some situations, it is necessary to allocate a region of memory within which particular fields will contain both pointer and non-pointer data. Such an object is allocated using the allocUnionArray() routine, prototyped below:

Detailed Description Text (508):
This routine allocates an object with the specified number of words and an accompanying signature within which all tags are initially set to indicate that fields contain non-pointers.

Detailed Description Text (509):
If the type of a particular word of this object must be changed at some later time to a pointer, its type tag is modified by using the setSigPtrTag() routine:

Detailed Description Text (511):
This routine sets the tag for the object at word.sub.-- offset positions from the start of obj to indicate that the corresponding word contains a pointer. As a side effect, this routine overwrites the corresponding word with NULL. If at some later time it is necessary to change the word from a pointer to a non-pointer, use the clrSigPtrTag() routine:

Detailed Description Text (515):
Slow versions of each routine are prototyped below. These slow functions pass pointer parameters and return pointer results on the C stack. Prior to preemption, the routine saves relevant pointers to slow pointer variables set aside on the PERC pointer stack for this purpose.

Detailed Description Text (521):
String and substring data is special in that we may have arrays of bytes that are shared by multiple overlapping strings. The bytes themselves are represented in a

block of memory known to the garbage collector as a String. The programmer represents each string using a String object. FIG. 7 shows string objects x and y, representing the strings "embedded" and "bed" respectively. The value field of each string object is a pointer to the actual string data. The offset field is the offset, measured in bytes, of the start of the string within the corresponding StringData buffer. The count field is the number of bytes in the string. Note that count represents bytes, even though Unicode strings might require two bytes to represent each character.

Detailed Description Text (530):
Slow versions of each of the routines described above are prototyped below. These slow functions pass pointer parameters and return pointer results on the C stack. Prior to preemption, the routine saves relevant pointers to slow pointer variables set aside on the PERC pointer stack for this purpose.

Detailed Description Text (543):
For objects residing in from-space which have been scheduled for copying into to-space, the Scan List field is overwritten with a pointer to the to-space copy. Otherwise, the Scan List field holds NULL.

Detailed Description Text (545):
b. The Indirect Pointer refers to the currently valid copy of the data that corresponds to this object. For objects in the mark and sweep region, this pointer always points to the object itself. For objects in to- and from-space, the pointer points to whichever version of the object currently represents the object's contents.

Detailed Description Text (546):
c. Activity Pointer points to the real-time activity object that was responsible for allocation of this object or has the NULL value if this object was not allocated by a real-time activity. When this object's memory is reclaimed, that real-time activity's memory allocation budget will be increased. Furthermore, if this object needs to be finalized when the garbage collector endeavors to collect it, the object will be placed on a list of this real-time activity's objects which are awaiting finalization. To distinguish objects that need to be finalized, the 0.times.01 bit (FINAL.sub.-- LINK) and the 0.times.02 bit (FINAL.sub.-- OBJ) of the Activity Pointer field are set when a finalizable object is allocated.

Detailed Description Text (547):
d. Signature Pointer points to a structure that represents the internal organization of the PERC data within the object. For objects requiring finalization, the Finalize Link field is not represented in the signature.

Detailed Description Text (548):
4. Free segments are doubly linked. The Indirect Pointer field is used as a forward link and the Signature Pointer field is used as the backward link. The size of the free segment, in words, is stored in the Activity Pointer field, representing an integer. Note that objects residing on a free list are distinguished by the special SCAN.sub.-- FREE value stored in their Scan List field.

Detailed Description Text (553):
In Java, programmers can specify an action to be performed when objects of certain types are reclaimed by the garbage collector. These actions are specified by including a non-empty finalize method in the class definition. Such objects are said to be finalizable. When a finalizable object is allocated, the two low order bits of the Activity Pointer are set to indicate that the object is finalizable. The 0.times.01 bit, known symbolically as FINAL.sub.-- LINK, signifies that this object has an extra Finalize Link field appended to the end of it. The 0.times.02 bit, known symbolically as FINAL.sub.-- OBJ, signifies that this object needs to be finalized. After the object has been finalized once, its FINAL.sub.-- OBJ is

cleared, but its FINAL.sub.-- LIINK bit remains on throughout the object's lifetime.

Detailed Description Text (554):
See FIG. 3 for an illustration of how finalization lists are organized. In this figure, Finalizees is a root pointer. This pointer refers to a list of finalization-list headers. There is one such list for each of the currently executing real-time activities, and there is one other list that represents all of the objects allocated by non-real-time activities. These lists are linked through the Activity Pointer field of the objects waiting to be finalized.

Detailed Description Text (555):
The run-time system includes a background finalizer thread which takes responsibility for incrementally executing the finalizers associated with all of the objects reachable from the Finalizees root pointer. Following execution of the finalizer method, the finalizes object is removed from the finalizes list and its Activity Pointer field is overwritten with a reference to the corresponding Activity object. Furthermore, we clear the FINAL.sub.-- OBJ so we don't finalize it again. Optionally, each real-time activity may take responsibility for timely finalization of its own finalizee objects. Typically, this is done within an ongoing real-time thread that is part of the activity's workload.

Detailed Description Text (556):
When an Activity object is first allocated, its pointer to the corresponding finalizee list head object is initialized to null. Later, when objects requiring finalization are encountered, a finalizee list head object is allocated and the Activity object's finalizee list head pointer is overwritten with a pointer to this object. Each time the activity's finalizee list becomes empty, we destroy the corresponding finalizee list head object, removing it from the Finalizees list, and overwrite the corresponding pointer within the Activity object with NULL.

Detailed Description Text (557):
The Finalize Link field is only present in objects that have finalization code. Throughout their lifetimes, all such objects have the FINAL.sub.-- LINK bit of the Activity Pointer field set at all times (and no objects that lack a Finalize Link field ever have this bit set). When first allocated, each finalizable object is linked through the Finalize Link field onto a single shared list (called the finalizable list) that represents all finalizable objects. When an object is recognized as ready for finalization, it is removed from the finalizable list and placed onto a finalizee list threaded through the Activity Pointer field.

Detailed Description Text (560):
1. Heap memory that has already been examined by the garbage collector must not be corrupted by writing into such heap objects pointers that have not yet been processed by the garbage collector. Otherwise, it might be possible for a pointer to escape scrutiny of the garbage collector. As a result, the referenced object might be treated as garbage and accidentally reclaimed. To avoid this problem, we impose a write barrier whenever pointers are written into the heap. (See the SetHeapPointer() macro in FIG. 41):

Detailed Description Text (561):
a. If the pointer to be written to memory refers to from-space, replace the pointer with the appropriate to-space address. Note that this may require that we set aside memory in to-space to hold the copy of the referenced from-space object.

Detailed Description Text (562):
b. If the pointer to be written to memory refers to a mark-and-sweep object that has not yet been marked, mark the object by placing it on the scan list.

Detailed Description Text (563):

2. We do not impose a read barrier. This means that pointers fetched from the internal fields of heap objects may refer to from-space objects or to mark-and-sweep objects that have not yet been marked. In case a pointer refers to a from-space object that has already been copied into to-space or to a to-space object that has not yet been copied into to-space, all references to heap object are indirected through the Indirection Pointer. (See FIG. 41)

Detailed Description Text (564):
3. Any objects that are newly allocated from the mark-and-sweep region have their Scan List pointer initialized to NULL. Thus, newly allocated objects will survive the current garbage collection pass only if pointers to these objects are written into the live heap.

Detailed Description Text (571):
2. Tend each root pointer. This consists of:

Detailed Description Text (572):
a. If the pointer refers to from-space, allocate space for a copy of this object in to-space and make the to-space copy's Indirect Pointer refer to the from-space object. Set the root pointer to refer to the to-space copy. Set the from-space copy's Scan List pointer to refer to the to-space copy.

Detailed Description Text (573):
b. Otherwise, if the pointer refers to the mark-and-sweep region or the to-space region and the referenced object has not yet been marked, mark the object. Marking consists of placing the object on the scan list. Each increment of garbage collection effort consists of the following:

Detailed Description Text (583):
7. Else, do a flip operation and restart the garbage collector. To-space and from-space are organized as illustrated in FIG. 4. In this illustration, live objects A, B, and C are being copied into to-space out of from-space. Objects B and C have been copied and object A is on the copy queue waiting to be copied. The arrows indicate the values of the Indirect Pointer fields in each of the invalid object copies. Memory to the right of the New pointer consists of objects that have been allocated during this pass of the garbage collector. Memory to the left of B' represents objects that were copied to to-space during the previous pass of the garbage collector. Garbage collection of the copy region consists of the following:

Detailed Description Text (585):
a. Atomically copy the object at position Relocated and update the from-space version of the object so that its Indirect Pointer refers to the to-space copy of the object.

Detailed Description Text (586):
b. As the object is being copied, tend any pointers that it might contain.

Detailed Description Text (587):
Additionally, tend the Activity Pointer field after masking out its two least significant bits and update the Signature Pointer if the signature is contained within this object. Scanning of the mark-and-sweep region consists of the following:

Detailed Description Text (589):
a. Scan the object at the head of the list. Scanning consists of tending each pointer contained within the object. Note that the scanner must scan the Activity Pointer field (after masking out the two least significant bits). A special technique is used to scan pointer stack objects. When pointer stack objects are scanned, the garbage collector consults the corresponding Thread object to

determine the current height of the pointer stack. Rather than scan the entire object, the garbage collector only scans that portion of the stack object that is currently being used.

Detailed Description Text (590):
b. Make the scan-list pointer refer to the next object on the scan list.

Detailed Description Text (591):
Scanning of PERC pointer stacks is special in the sense that only the portion of the stack that is live is scanned. Memory within the object that is above the top-of-stack pointer is ignored. In order to support this capability, PERC pointer stacks refer to their corresponding Thread object, enabling the garbage collector to consult the thread's top-of-stack pointer before scanning the stack object.

Detailed Description Text (595):
We remove the object from the finalizable list and place it (the newly created to-space copy if the object was originally found in from-space) onto a temporary holding list of finalizees threaded through the Finalize Link field. In order to support the remove operation, the scanning process maintains at all times a pointer to the preceding object on the finalizable list. Additionally, we mark this object by placing it on the scan list if the object resides in the mark-and-sweep region.

Detailed Description Text (598):
3. Wait for the scanning and copying process to complete. It is not necessary to rescan the root pointers because all of the objects now being scanned and copied are considered to be dead insofar as the application code is concerned. Thus, there is no possible way for a pointer to one of these "dead" objects to find its way into a root pointer.

Detailed Description Text (599):
4. Now, process the holding list of finalizees that was created in step 1, linking each finalizes onto the appropriate activity's finalizes list (or onto the Orphaned Finalizees list). This list is threaded through the Activity pointer field of the object's header. At this time, overwrite the object's Finalize Link field with NULL. If the activity to which an object corresponds does not currently have a finalizes list, it will be necessary in this step to allocate and initialize the finalizee list head. (See FIG. 3)

Detailed Description Text (606):
The final step is to zero out the old from-space so that future allocations from this region can be assumed to contain only zeros. Simply walk through memory from low to high address and overwrite each word with a zero. For each object encountered in from-space, we ask whether it was copied into to-space (by examining its Indirect Pointer). If it was not copied, we check to see if it is a HashLock object with a hash value that needs to be reclaimed. If so, we reclaim the hash value as described above, except that a new HashCache object may need to be allocated to represent the recycled hash value if there are no available slots in the existing list of recycled hash values. We allocate this HashCache object using the standard heap-memory allocator. Otherwise, we update the corresponding activity's tally that represents the total amount of this activity's previously allocated memory that has been garbage collected.

Detailed Description Text (608):
The standard model for execution of Java byte-code programs assumes an execution model comprised of a single stack. Furthermore, the Java byte codes are designed to support dynamic loading and linking. This requires the use of symbolic references to external symbols. Resolving these symbolic references is a fairly costly operation which should not be performed each time an external reference is accessed. Instead, the PERC virtual machine replaces symbolic references with more efficient integer index and direct pointer references when the code is loaded.

Detailed Description Text (609):
In order to achieve good performance, the PERC virtual machine does not check for type correctness of arguments each time it executes a byte-code instruction. Rather, it assumes that the supplied arguments are of the appropriate type. Since byte-code programs may be downloaded from remote computer systems, some of which are not necessarily trustworthy, it is necessary for the PERC virtual machine to scrutinize the byte-code program for type correctness before it begins to execute. The process of guaranteeing that all of the operands supplied to each byte-code instruction are of the appropriate type is known as byte code verification. Once the types of each operation are known, it is possible to perform certain code transformations. Some of these transformations are designed simply to improve performance. In other cases, the transformations are needed to comply with the special requirements of the PERC virtual machine's stack protocols. For example, Java's dup2 byte code duplicates the top two elements on the Java stack. Byte-code verification determines the types of the top two stack elements. If both are of type pointer, the class loader replaces this byte code with a special instruction named dup2.sub.-- 11, which duplicates the top two elements of the pointer stack. If the two stack arguments are both non-pointer values, the PERC class loader replaces this byte code with the dup2.sub.-- 00 instruction, which duplicates the top two elements of the non-pointer stack. If one of dup's stack arguments is a pointer and the other is a non-pointer (in either order), the PERC class loader replaces dup with dup2.sub.-- 10, which duplicates the top element on each stack. A complete list of all the transformations that are performed by the byte code loader is provided in the remainder of this section.

Detailed Description Text (616):
2. A list of pointers to the basic block objects that may branch to this block. We call these blocks the predecessors.

Detailed Description Text (617):
3. A list of pointers to the basic block objects that this block may branch to. We call these blocks the successors.

Detailed Description Text (633):
item found on the specified one-byte indexed position within the constant pool table onto the stack. If this item is an object pointer, we need to push the pointer value onto the pointer stack. If this item is not a pointer, we push its value onto the non-pointer stack. We use code 18 to represent ldc1.sub.-- np, which loads a non-pointer constant onto the non-pointer stack. We use code 255 to represent ldc1.sub.-- p, which loads a pointer constant onto the pointer stack. ldc2. This operation is represented by code 19. This instruction pushes the item found on the specified two-byte indexed position within the constant pool table onto the stack. If this item is an object pointer, we need to push its value onto the pointer stack. If this item is not a pointer, we push its value onto the non-pointer stack. We use code 19 to represent ldc2.sub.-- np, which loads a non-pointer constant onto the non-pointer stack. We use code 254 to represent ldc2.sub.-- p, which loads a pointer constant onto the pointer stack.

Detailed Description Text (635):
1. putfield.sub.-- q encoded as 181: We replace the constant-pool entry with an integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less than 64 Kbytes and can thus be represented in the 16-bit immediate operand

representing an unsigned integer quantity.

Detailed Description Text (636):
2. putfield.sub.-- qnp8 encoded as 253: We replace the two-byte immediate operand with the offset of the 8-bit non-pointer field within the corresponding object.

Detailed Description Text (637):
3. putfield.sub.-- qnp16 encoded as 252: We replace the two-byte immediate operand with the offset of the 16-bit non-pointer field within the corresponding object.

Detailed Description Text (638):
4. putfield.sub.-- qnp32 encoded as 251: We replace the two-byte immediate operand with the offset of the 32-bit non-pointer field within the corresponding object.

Detailed Description Text (639):
5. putfield.sub.-- qnp64 encoded as 250: We replace the two-byte immediate operand with the offset of the 64-bit non-pointer field within the corresponding object.

Detailed Description Text (640):
6. putfield.sub.-- qp encoded as 249: We replace the two-byte immediate operand with the offset of the 32-bit pointer field within the corresponding object.

Detailed Description Text (642):
1. getfield.sub.-- q encoded as 180: We replace the constant-pool entry with a 32-bit integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less than 64 Kbytes and can thus be represented in the 16-bit immediate operand representing an unsigned integer quantity.

Detailed Description Text (643):
2. getfield.sub.-- qnp8 encoded as 248: We replace the two-byte immediate operand with the offset of the 8-bit non-pointer field within the corresponding object.

Detailed Description Text (644):
3. getfield.sub.-- qnp16 encoded as 247: We replace the two-byte immediate operand with the offset of the 16-bit non-pointer field within the corresponding object.

Detailed Description Text (645):
4. getfield.sub.-- qnp32 encoded as 246: We replace the two-byte immediate operand with the offset of the 32-bit non-pointer field within the corresponding object.

Detailed Description Text (646):
5. getfield.sub.-- qnp64 encoded as 245: We replace the two-byte immediate operand with the offset of the 64-bit non-pointer field within the corresponding object.

Detailed Description Text (647):
6. getfield.sub.-- qp encoded as 244: We replace the two-byte immediate operand with the offset of the 32-bit pointer field within the corresponding object.

Detailed Description Text (648):
Putstatic. This operation is represented by code 179. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a

pointer to the Field structure that describes the field to be updated. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (651):
3. putstatic.sub.-- qnp32 encoded as 241 if the field is a 32-bit non-pointer.

Detailed Description Text (652):
4. putstatic.sub.-- qnp64 encoded as 240 if the field is a 64-bit non-pointer.

Detailed Description Text (653):
5. putstatic.sub.-- qp encoded as 239 if the field is a 32-bit pointer.

Detailed Description Text (654):
Getstatic. This operation is represented by code 178. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a pointer to the Field structure that describes the field to be fetched. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (657):
3. getstatic.sub.-- qnp32 encoded as 236 if the field is a 32-bit non-pointer.

Detailed Description Text (658):
4. getstatic.sub.-- qnp64 encoded as 235 if the field is a 64-bit non-pointer.

Detailed Description Text (659):
5. getstatic.sub.-- qp encoded as 234 if the field is a 32-bit pointer.

Detailed Description Text (660):
Anewarray. This operation is represented by code 189. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool. When resolved, the selected constant must be a class. The result of this operation is a newly allocated array of pointers to the specified class. The loader replaces this instruction with anewarray.sub.-- q, which is also encoded as operation 189. This instruction differs from anewarray in that it does not need to resolve the constant entry. Rather, it assumes that the specified slot of the constant pool holds a pointer directly to the corresponding class object.

Detailed Description Text (661):
Multianewarray. This operation is represented by code 197. It takes two immediate-mode byte operands to represent a 16-bit constant pool index and a third immediate-mode byte operand to represent the number of dimensions in the array to be allocated. The index position is handled the same as for anewarray. The loader replaces this instruction with multianewarray.sub.-- q, which is encoded as operation 197. This instruction differs from multianewarray in that it does not need to resolve the constant entry. Rather, it assumes that the specified slot of the constant pool holds a pointer directly to the corresponding class object.

Detailed Description Text (662):
Invokevirtual. This operation is represented by code 182. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method signature, including its name. If the method-table index of the corresponding method is greater than 255 or if the number of pointer arguments is greater than 255, the loader replaces this instruction with invokevirtual.sub.-- q, encoded as operation 182. Otherwise, the

loader replaces this instruction with invokevirtual.sub.-- fq, encoded as operation 233.

Detailed Description Text (663):
With the invokevirtual.sub.-- fq instruction, the first immediate-mode byte operand represents the method table index and the second immediate-mode byte operand represents the number of pointer arguments.

Detailed Description Text (664):
With the invokevirtual.sub.-- q instruction, the two immediate-mode operands represent the same 16-bit index into the constant pool table as with the original invokevirtual instruction. However, this entry within the constant pool table is overwritten with a pointer to the Method structure that describes this method. (Note that both invokevirtual and invokespecial may share access to this same entry in the constant pool. In fact, there is no difference between the implementations of invokespecial.sub.-- q and invokestatic.sub.-- q in certain frameworks.)

Detailed Description Text (665):
invokespecial. This operation is represented by code 183. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method signature, including its name. This instruction is replaced with invokespecial.sub.-- q, encoded as 183. With the invokespecial.sub.-- q instruction, the selected constant pool cntry is replaced with a pointer to the Method structure that describes this method. (Note that both invokevirtual and invokespecial may share access to this same entry in the constant pool.)

Detailed Description Text (666):
invokestatic. This operation is represented by code 184. It takes two immediate-mode byte operands which represent a 16-bit index into the constant pool table. The corresponding constant-pool entry represents the method's class and signature, including its name. The loader replaces this instruction with invokestatic.sub.-- q, encoded as 184. The distinction of invokestatic.sub.-- q is that the selected constant pool entry is a pointer to the Method structure that describes this method.

Detailed Description Text (667):
Invokeinterface. This operation is represented by code 185. The instruction takes a 2-byte constant pool index, a one-byte representation of the number of arguments, and a one-byte reserved quantity as immediate-mode operands. The corresponding constant-pool entry represents the method's signature. The loader replaces this instruction with invokeinterface.sub.-- q, encoded as 185. The distinction of invokeinterface.sub.-- q is that the constant pool entry is overwritten with a pointer to a Method structure that represents the name and signature of the interface method and the reserved operand is overwritten with a guess suggesting the "most likely" slot at which the invoked object's method table is likely to match the invoked interface. If this slot does not match, this instruction searches the object's method table for the first method that does match. On each execution of invokeinterface.sub.-- q, the guess field is overwritten with the slot that matched on the previous execution of this instruction.

Detailed Description Text (669):
New. This operation is represented by code 187. The instruction takes a 2-byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with new.sub.-- q, also encoded as 187. The distinction of new.sub.-- q is that the constant pool entry is replaced with a pointer to the resolved class object.

Detailed Description Text (670):
Checkcast. This operation is represented by code 192. The instruction takes a 2-

byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with checkcast.sub.-- q, also encoded as 192. The distinction of checkcast.sub.-- q is that the constant pool entry is replaced with a pointer to the resolved class object.

Detailed Description Text (671):
Instanceof. This operation is represented by code 193. The instruction takes a 2-byte constant pool index. The constant pool entry is a class name that can be resolved to a class pointer. The loader replaces this instruction with instanceof.sub.-- q, also encoded as 193. The distinction of instanceof.sub.-- q is that the constant pool entry is known to have been replaced with a pointer to the resolved class object.

Detailed Description Text (673):
The standard Java byte code assumes that all local variables and all push and pop operations refer to a single shared stack. Offsets for local variables are all calculated based on this assumption. Our implementation maintains two stacks, one for non-pointers and another for pointers. Pointer local variables are stored on the pointer stack. And non-pointer locals are stored on the non-pointer stack. Thus, our byte-code loader has to remap the offsets for all local variable operations. The affected instructions are: iload, iload.sub.-- <n>, lload, lload.sub.-- <n>, fload, fload.sub.-- <n>, dload, dload.sub.-- <n>, aload, aload.sub.-- <n>, istore, istore.sub.-- <n>, lstore, lstore.sub.-- <n>, fstore, fstore.sub.-- <n>, dstore, dstore.sub.-- <n>, astore, astore.sub.-- <n>, iinc.

Detailed Description Text (676):
We want to make sure that operations that manipulate the stack are properly configured to differentiate between the pointer stack and the non-pointer stack.

Detailed Description Text (677):
Pop. This operation is represented by code 87. It removes the top item from the stack. We usc code 87 to represent pop.sub.-- 0, which pops from the non-pointer stack, and code 232 to represent pop.sub.-- 1, which pops from the pointer stack.

Detailed Description Text (678):
pop2. This operation is represented by code 88. It removes the top two items from the stack. We use code 88 to represent pop2.sub.-- 00, which pops two values from the non-pointer stack, code 231 to represent pop2.sub.-- 10 which pops one value from each stack, and code 230 to represent pop2.sub.-- 11, which pops two values from the pointer stack.

Detailed Description Text (679):
Dup. This operation is represented by code 89. It duplicates the top stack item. We use code 89 to represent dup.sub.-- 0, which duplicates the top non-pointer stack entry, and code 229 to represent dup.sub.-- 1, which duplicates the top pointer stack entry.

Detailed Description Text (680):
dup2. This operation is represented by code 92. It duplicates the top two stack items. We use code 92 to represent dup.sub.2.sub.-- 00, which duplicates the top two non-pointer stack entries, code 228 to represent dup2.sub.-- 10, which duplicates the top entry on each stack, and code 227 to represent dup2.sub.-- 11, which duplicates the top two pointer stack entries.

Detailed Description Text (681):
dup .times.1. This operation is represented by code 90. It duplicates the top stack item, shifts the top two stack items up one position on the stack, and inserts the duplicated top stack item into the newly vacated stack position. Note that the translation of this instruction depends on the types of the top two stack values at

the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 01 represents the condition in which the top stack element is a pointer and the next entry is a non-pointer. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (682):
00: We use code 90 to represent dup.sub.-- .times.1.sub.-- 00, which does its manipulations entirely on the non-pointer stack.

Detailed Description Text (683):
01: Reuse code 229 to represent dup.sub.-- 1, which duplicates only a single pointer value (this is the appropriate action to perform when the top stack element is a pointer, and the second element is a non-pointer).

Detailed Description Text (684):
10: Reuse code 89 to represent dup.sub.-- 0, which duplicates only a single non-pointer value (this is the appropriate action to perform when the top stack element is a non-pointer and the second element is a pointer).

Detailed Description Text (685):
11: Use code 226 to represent dup.sub.-- .times.1.sub.-- 11, which does all of its manipulations on the pointer stack.

Detailed Description Text (686):
dup .times.2. This operation, encoded as 91, duplicates the top stack entry, shifts the top three stack entries up one stack position, and inserts the duplicated stack entry into the newly vacated stack position. Note that the translation of this instruction depends on the types of the top three stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 001 represents the condition in which the top stack element is a pointer and the next two entries are non-pointers. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (687):
000: We use code 91 to represent dup.sub.-- .times.2.sub.-- 000, which does its manipulations entirely on the non-pointer stack.

Detailed Description Text (688):
001: Reuse code 229 to represent dup.sub.-- 1, which duplicates only a single pointer value (this is the appropriate action to perform when the top stack element is a pointer, and the next two elements are non-pointers).

Detailed Description Text (689):
010: Reuse code 90 to represent dup.sub.-- .times.1.sub.-- 00, which duplicates the non-pointer value and inserts it into the appropriate position on the non-pointer stack.

Detailed Description Text (690):
011: Reuse code 226 to represent dup.sub.-- .times.1.sub.-- 11, which duplicates the pointer value and inserts it into the appropriate position on the pointer stack.

Detailed Description Text (691):
100: Reuse code 90 to represent dup.sub.-- .times.1.sub.-- 00, which duplicates the

non-pointer value and inserts it into the appropriate position on the non-pointer stack.

Detailed Description Text (692):
101: Reuse code 226 to represent dup.sub.-- .times.1.sub.-- 11, which duplicates the pointer value and inserts it into the appropriate position on the pointer stack.

Detailed Description Text (693):
110: Reuse code 89 to represent dup.sub.-- 0, which duplicates only a single non-pointer value (this is the appropriate action to perform when the top stack element is a non-pointer and the second element is a pointer).

Detailed Description Text (694):
111: We use code 225 to represent dup.sub.-- '2.sub.-- 111, which does its manipulations entirely on the pointer stack.

Detailed Description Text (695):
dup2 .times.1. This operation, encoded as 93, duplicates the top two stack entries, shifts the top three stack entries up two stack positions, and inserts the duplicated stack entries into the newly vacated stack slots. Note that the translation of this instruction depends on the types of the top three stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 001 represents the condition in which the top stack element is a pointer and the next two entries are non-pointers. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (696):
000: We use code 93 to represent dup2.sub.-- .times.1.sub.-- 000, which does its manipulations entirely on the non-pointer stack.

Detailed Description Text (699):
011: Reuse code 227 to represent dup2.sub.-- 11, which duplicates the top two values on the pointer stack.

Detailed Description Text (700):
100: Reuse code 92 to represent dup2.sub.-- 00, which duplicates the top two values on the non-pointer stack.

Detailed Description Text (703):
111: We use code 222 to represent dup2.sub.-- .times.1.sub.-- 111, which does its manipulations entirely on the pointer stack.

Detailed Description Text (704):
dup2 .times.2. This operation is represented by code 94. It duplicates the top two stack items, shifts the top four stack items up two positions on the stack, and inserts the duplicated stack items into the newly vacated stack positions. Note that the translation of this instruction depends on the types of the top four stack values at the time this instruction is executed. Each stack entry is either a pointer or a non-pointer. Use a binary 1 to represent pointers and a binary 0 to represent non-pointers. Assemble the type codes from left to right, with the top stack entry being represented by the right-most binary digit. Thus, the number 0001 represents the condition in which the top stack element is a pointer and the next three are non-pointers. Each combination of four binary digit type codes represents a decimal number. We characterize the handling given to each case as tagged by the binary representation of the stack type codes:

Detailed Description Text (705):
0000: We use code 94 to represent dup2.sub.-- .times.2.sub.-- 0000, which does all its manipulations on the non-pointer stack.

Detailed Description Text (708):
0011: We reuse the code for dup2.sub.-- 11 (which is the right thing to do if the top two stack elements are pointers, and the next two are non-pointers).

Detailed Description Text (709):
0100: We reuse code for dup2.sub.-- .times.1.sub.-- 000. This instruction duplicates the top two entries on the non-pointer stack, shifts the top three entries of the non-pointer stack up two positions, and inserts the duplicated values into the vacated stack slots.

Detailed Description Text (712):
0111: We reuse the code for dup2.sub.-- .times.1.sub.-- 111. This instruction duplicates the top two entries of the pointer stack, shifts the top three values of the pointer stack up two positions on the stack, and inserts the duplicated pointer values into the vacated stack position.

Detailed Description Text (720):
1111: We use code 218 to represent dup2.sub.-- .times.2.sub.-- 1111. This instruction duplicates the top two entries on the pointer stack, shifts the top four entries on the pointer stack up two positions, and inserts the duplicated pointer values into the newly vacated pointer slots.

Detailed Description Text (722):
00: We use code 95 to represent swap.sub.-- 00, which exchanges the top two values on the non-pointer stack.

Detailed Description Text (725):
11: We use code 217 to represent swap.sub.-- 11, which exchanges the top two values on the pointer stack.

Detailed Description Text (727):
JIT-generated methods usc only the PERC pointer and non-pointcr stacks. All pointer information is stored on the pointer stack and all non-pointer information is stored on the non-pointer stack. The non-pointer activation frame is illustrated in FIG. 94. The pointer activation frame is identical except that there is no return address stored in the pointer activation frame.

Detailed Description Text (728):
Within a JIT-generated method, all local variables, including incoming and outgoing arguments are referenced at fixed offsets from the register that represents the corresponding stack pointer. There is no need for a frame pointer because the stack pointer remains constant throughout execution of the method.

Detailed Description Text (729):
Note that the JIT method's prologue subtracts a constant value from the stack pointer and the JIT method's epilogue adds the same constant value to the stack pointer.

Detailed Description Text (730):
When JIT-compiled methods invoke byte-code or native-codc methods, the corresponding byte-code stub sets up the frame and stack pointers necessary for execution of the corresponding C routines. Additionally, the return address is removed from the non-pointer stack and stored temporarily in a C local variable within the stub procedure.

Detailed Description Text (731):

Within a JIT-compiled method, machine registers are partitioned so that certain registers are known to only contain base pointers and other machine registers are known to only contain non-pointers. An additional class of registers may contain derived pointers which refer to the internal fields of particular objects. Each derived-pointer register is always paired with a base-pointer register which is known to identify the starting address of the corresponding object. Otherwise, the derived-pointer register holds the NULL value.

Detailed Description Text (741):
2. All Indirect Pointers are initialized to refer to the object itself. This enables standard heap-access macros to work correctly when referring to ROM objects.

Detailed Description Text (744):
a. For ROM preloading, the representation of a class data structure is simply a template which will be copied into RAM when the system actually executes. Since the executing application code must be able to modify the class's static variables, the class representation's static variable pointer points to a signature (represented within the load file) which is used by the bootstrap "loader" to allocate the class's static variable structure.

Detailed Description Paragraph Table (3):
Field Name Field Size Description
Object.sub.-- Size 32 bits This represents the total number of words in the object region (Object.sub.-- Region).
Relocatable.sub.-- M (Object.sub.-- Siz This field maintains 1 bit for each word of ap e/32) words the object region. The bit is on if the (rounded corresponding word holds a non-null up) pointer and is off otherwise. All non-null pointers are assumed to point within the object region. The first word of the object region (Object.sub.-- Region) is represented by bit 0x01 of the first word of the Relocatable.sub.-- Map. Class.sub.-- Table 32 bits This field represents the offset within the object region (Object.sub.-- Region) of the table that represents all of the classes defined by this object. Object.sub.-- Region Object.sub.-- Size This represents the ROM memory image. words Each object in this memory region is provided with a standard garbage collection header (See Figure 2), including a Scan List pointer, Indirect Pointer, Activity Pointer, and Signature Pointer. In the ROM image, all pointers, including the pointers stored within object headers, are represented by offsets relative to the beginning of the Object.sub.-- Region. All objects are initialized to belong to the 0 Activity.

CLAIMS:

1. A real-time virtual machine method (RTVMM) for implementing real-time systems and activities, the RTVMM comprising the steps:

implementing an O-OPL program that can run on computer systems of different designs, an O-OPL program being based on an object-oriented programming language (O-OPL) comprising object type declarations called classes, each class definition describing the variables that are associated with each object of the corresponding class and all of the operations called methods that can be applied to instantiated objects of the specified type, a "method" being a term of art describing the unit of procedural abstraction in an object-oriented programming system, an O-OPL program comprising one or more threads wherein the run-time stack for each thread is organized so as to allow accurate identification of type-tagged pointers contained on the stack without requiring type tag information to be updated each time the stack's content changes, the O-OPL being an extension of a high-level language (HLL) exemplified by Java, HLL being an extension of a low-level language (LLL) exemplified by C and C++, a thread being a term of art for an independently-executing task, an O-OPL program being represented at run time by either O-OPL byte

codes or by native machine codes.

2. The RTVMM of claim 1 wherein an O-OPL program utilizes a pointer stack and a non-pointer stack.

12. The RTVMM of claim 1 wherein one of the implemented threads is a garbage collection thread that operates asynchronously thereby resulting in the garbage collection thread being interleaved with other threads in arbitrary order, objects subject to garbage collection being either finalizable or non-finalizable, a finalizable object being subject to an action that is performed when the memory space allocated to the finalizable object is reclaimed by the garbage collection thread, the finalizing action being specified by including a non-empty finalizer method in the class definition, the garbage collection thread being able to distinguish a thread's pointer variables from the thread's non-pointer variables, preemption of a thread being allowed only if the thread is in a state identified as a preemption point, a thread being allowed to hold pointers in variables between preemption points that may not be visible to the garbage collection thread, pointer variables that may not be visible to the garbage collection thread being called fast pointers, pointer variables that are visible to the garbage collection thread being called slow pointers, each LLL, function being identified as either preemptible or non-preemptible.

13. The RTVMM of claim 12 wherein the implementing step comprises the steps:

causing the values of essential fast pointers to be copied into slow pointers immediately prior to a preemption point of a preemptible thread. (5.0/4)

causing the values of essential fast pointers to be restored after preemption by causing the values of the slow pointers to be copied to the locations where the values of the fast pointers were previously stored.

14. The RTVMM of claim 12 wherein the implementing step comprises the steps:

causing the values of all of the essential fast pointers of a preemptible LLL function to be copied into slow pointers prior to calling the prcemptible LLL function;

causing the values of the essential fast pointers to be restored when the called preemptible LLL function returns by causing the values of the slow pointers to be copied to the locations where the values of the fast pointers were previously stored.

17. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that returns the value of a fast pointer in the heap given the identity of the pointer and its type.

18. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that assigns a value from a fast pointer in heap memory given the identity of the pointer, its type, and the value.

21. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing direct access to stack data using LLL pointer indirection.

22. The RTVMM of claim 21 wherein the implementing step comprises the step:

representing stack pointers by LLL global variables declared as pointers.

25. The RTVMM of claim 24 wherein the implementing step comprises the step:

including an "activity pointer" field for each object in memory, the "activity pointer" identifying the activity that was responsible for allocating the object, the "activity pointer" field containing a "null" value if the object was not allocated by a real-time activity.

26. The RTVMM of claim 25 wherein the implementing step comprises the step:

maintaining a free pool of space segments for to-space and for each mark-and-sweep sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "activity pointer" field to specify the size of a free space segment.

27. The RTVMM of claim 24 wherein the implementing step comprises the step:

including a "signature pointer" field for each object in memory, the "signature pointer" field containing a pointer to a structure that represents the internal organization of the O-OPL data within the object.

28. The RTVMM of claim 27 wherein the implementing step comprises the steps:

maintaining a free pool of space segments for to-space and for each mark-and-sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "signature pointer" field to be used as a backward link to the preceding segment..

30. The RTVMM of claim 24 wherein the implementing step comprises the step:

including a "scan list" field for each object in memory, the "scan list" field distinguishing marked and unmarked objects residing in a mark-and-sweep space but not on a free list, the "scan list" field for each object in a mark-and-sweep space having a "scan clear" value at the beginning of a garbage collection cycle, an object recognized as being a live object being placed on a list of recognized live objects, the "scan list" field for an object on the list of recognized live objects having either a "scan end" value denoting the last object on the list of recognized live objects or a value identifying the next object on the list of recognized live objects, the "scan list" field for an object residing on a free list within a mark-and-sweep space or to- space having the "scan free" value, the "scan list" field for an object residing in from-space which has been scheduled for copying into to-space being a pointer to the to-space copy, the "scan list" field otherwise being assigned the "scan clear" value, the "scan list" field for an object residing in to-space having the "scan clear" value at the beginning of a garbage collection cycle, a to-space object recognized as live during garbage collection being placed on a list of recognized live objects, the "scan list" field for a to-space object on the list of recognized live objects having a value identifying the next object on the list of recognized live objects, the "scan list" field for each object queued for copying into to-space having the "scan end" value denoting that the object is live.

34. The RTVMM of claim 24 wherein the implementing step comprises the steps:

causing a "finalize link" bit and a "finalize object" bit in an "activity pointer"

field of a finalizable object to be set when space is allocated to the finalizable object, the "finalize link" bit being set indicating that the object has a "finalize link" field appended to the object, the "finalize object" bit being set indicating that the object needs to be finalized;

causing the "finalize object" bit to be cleared when a finalizable object has been finalized.

35. The RTVMM of claim 24 wherein a pointer is to be written into memory, the implementing step comprising the steps: causing the pointer to an object in from-space to be replaced by a pointer to the object's new address in to-space;

causing an object in mark-and-sweep space to which the pointer points to be marked if the object has not yet been marked.

38. The RTVMM of claim 24 wherein the implementing step comprises the steps:

maintaining a list of root pointers to live objects;

causing space for a copy of an object in to-space to be allocated if a root pointer to the object refers to from-space;

causing the from-space address of the object to be written in an "indirect pointer" field of the object's allocated space in to-space;

causing the root pointer to be replaced with the address of the object in to-space;

causing the to-space address of the object to be written into a "scan list" field of the object in from-space.

39. The RTVMM of claim 24 wherein the implementing step comprises the steps:

maintaining a list of root pointers to live objects;

causing an object to be marked if the root pointer to the object refers to a mark-and-sweep space or to-space and the object has not yet been marked, marking consisting of placing the object on a scan list.

43. The RTVMM of claim 40 wherein the transfer of objects needing finalization on the list of finalizable objects to the finalizee list has been completed, the implementing step comprising the steps:

causing the objects on the copy list to be copied to to-space;

causing the objects on the scan list to be scanned, scanning consisting of tending each pointer contained within an object, tending being a term of art describing the garbage collection process of (1) examining a pointer and, if the object has not already been recognized as live, arranging for the referenced object to be subsequently scanned by placing the object on a scan list if it resides in a mark-and-sweep space or in to-space or by arranging for the object to be copied into to-space if it resides in from-space and (2) updating the pointer to refer to the object's new location if it has been queued for copying into to-space.

45. The RTVMM of claim 44 wherein an activity's finalizee list is implemented by placing in an "activity pointer" field of a finalizee the address of the next finalizes on the activity's finalizee list.

46. The RTVMM of claim 44 wherein after transferring a finalizee on the finalizee list to the appropriate activity's finalizee list or onto an orphaned finalizee

list, the implementing step comprises the step:

causing a "finalize link" bit in an "activity pointer" field of the object corresponding to the finalizee to be cleared, a cleared "finalize link" bit indicating that the object is no longer on the list of finalizable objects.

51. The RTVMM of claim 12 wherein the implementing step comprises the step:

designating portions of memory as a to-space, from-space, and zero or more mark-and-sweep spaces;

including an "indirect pointer" field for each object in memory, the "indirect pointer" field containing a pointer to the location of the currently valid copy of the data that corresponds to the object, the pointer pointing to the object itself for objects in a mark-and-sweep space, the pointer pointing to the location of the object that currently represents the object's contents for objects in to-space and from-space.

52. The RTVMM of claim 51 wherein the implementing step comprises the steps:

maintaining a free pool of space segments for to-space and for each mark-and-sweep space, a free pool being organized as a plurality of doubly-linked lists, each linked list being a list of free space segments ranging in size from a lower value to an upper value, the size ranges for the plurality of linked lists being non-overlapping;

causing the "indirect pointer" field to be used as a forward link to the succeeding segment.

53. The RTVMM of claim 12 wherein the implementing step comprises the step:

including an "activity pointer" field for each object in memory, the "activity pointer" identifying the real-time activity object that was responsible for allocation of the object, the "activity pointer" field containing a "null" value if the object was not allocated by a real-time activity;

maintaining a finalizees list of objects waiting to be finalized for each real-time activity, the objects on the finalizees list being linked through the "activity pointer" field;

maintaining a list of the headers of the finalizees lists, the pointer "finalizees" being a root pointer to the headers list.

54. The RTVMM of claim 53 wherein the implementing step comprises the step:

implementing a finalizer thread that operates in the background and is

responsible for incrementally executing the finalizer methods associated with finalizee objects reachable from the "finalizees" pointer.

55. The RTVMM of claim 54 wherein the finalizer thread comprises the steps:

causing a finalizer method associated with a finalizee object to be executed;

causing the finalizee object to be removed from the associated finalizee list;

causing the "activity pointer" field of the finalizee object to be overwritten with a reference to the allocating object;

causing a "finalize object" bit in the "activity pointer" field of the finalizee

object to be cleared indicating that the object has been finalized.

56. The RTVMM of claim 53 wherein the implementing step comprises the step:

implementing a finalizer thread that is part of a real-time activity and is responsible for incrementally executing the finalizer methods associated with finalizee objects associated with the real-time activity and reachable from the "finalizees" pointer.

57. The RTVMM of claim 56 wherein the finalizer thread comprises the steps:

causing a finalizer method associated with a finalizee object to be executed;

causing the finalizee object to be removed from the associated finalizee list;

causing the "activity pointer" field of the finalizee object to be overwritten with a reference to the allocating object;

causing a "finalize object" bit in the "activity pointer" field of the finalizee object to be cleared indicating that the object has been finalized.

58. The RTVMM of claim 53 wherein the implementing step of claim 1 comprises the steps:

causing memory space to be allocated to a finalizee list head object when an object associated with a particular activity and requiring finalization is encountered;

causing a finalizee list head pointer associated with the activity to be overwritten with a pointer to the finalizee list head object;

causing the finalizee list head object to be destroyed when the finalizee list becomes empty and overwriting the finalizee list head pointer with the "null" value.

72. The RTVMM of claim 1 wherein the implementing step comprises the step:

causing symbolic references to be replaced with integer indices and direct pointer references when a program is loaded into a computer system.

77. The RTVMM of claim 1 wherein an O-OPL byte-code loader is used to load an HLL byte-code program into a computer system, the implementing step comprises the step:

causing each byte code to be examined when a class is loaded to determine whether it operates on pointer or non-pointer data;

causing pointers to be pushed onto and popped from a pointer stack;

causing non-pointers to bc pushed onto and popped from a non-pointer stack.

79. The RTVMM of claim 1 wherein the implementing step comprises the step:

utilizing only O-OPL pointer and non-pointer stacks in executing methods compiled by a JIT compiler, JIT standing for "just in time" and denoting a process for translating HLL, byte codes to native machine language codes on the fly, just in time for its execution, the translation of byte codes to native codes being a form of JIT compiling.

80. The RTVMM of claim 79 wherein a method compiled by a JIT compiler invokes a byte-code or nativc-code method, the implementing step comprises the step:

causing the frame and stack pointers necessary for the execution of the corresponding LLL routines to be set up;

causing the return address to be removed from the non-pointer stack and stored temporarily in an LLL local variable.

84. The RTVMM of claim 1 wherein each thread maintains its own versions of global variables "pointer stack pointer" (psp), "pointer stack frame pointer" (pfp), "non-pointer stack pointer" (npsp) and "non-pointer stack frame pointer" (npfp), the implementing step comprising the steps:

creating a thread called a thread dispatcher, the thread dispatcher saving psp, pfp, npsp, and npfp into the state variables of an executing thread when the executing thread is preempted, the preempted thread restoring these state variables when the preempted thread resumes execution.

87. The RTVMM of claim 85 wherein the implementing step relating to the load file includes the step:

causing the "indirect pointer" field of each object to refer to itself.

First Hit Fwd Refs
End of Result Set

☐ Generate Collection Print

L30: Entry 1 of 1 File: USPT Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A
TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Detailed Description Text (630):
Most of the operations that access the constant pool can be replaced with fast variants. When a Java class is loaded into the Java virtual machine, all of the constants associated with each method are loaded into a data structure known as the constant pool. Because Java programs are linked together at run time, many constants are represented symbolically in the byte code. Once the program has been loaded, the symbolic values are replaced in the constant pool with the actual constants they represent. We call this process "resolving constants." Sun Microsystems Inc.'s descriptions of their Java implementation suggest that constants should be resolved on the fly: each constant is resolved the first time it is accessed by user code. Sun Microsystems Inc.'s documents further suggest that once an instruction making reference to a constant value has been executed and the corresponding constant has been resolved, that byte code instruction should be replaced with a quick variant of the same instruction. The main difference between the quick variant and the original instruction is that the quick variant knows that the corresponding constant has already been resolved.

Detailed Description Text (749):
1. The code used in the implementation of the ROMizer tool to read in a Java class file, verify the validity of the byte code, and transform the byte code into the PERC instruction set is the exact same code that is used by the PERC implementation to support dynamic (on-the-fly) loading of new byte-code functionality into the PERC virtual machine.

L8: Entry 5 of 11 File: USPT Aug 13, 2002

DOCUMENT-IDENTIFIER: US 6434694 B1
TITLE: Security for platform-independent device drivers

Abstract Text (1):
A computer-implemented method for allocating securely memory resources to a platform-independent device driver is described. In one embodiment, a device driver generates a request for memory allocation in terms of an abstract memory address space. The driver forwards the request to the bus manager. An inner class representation of the bus manager is generated and the identity of the driver is determined. The inner class representation handles the request from the device driver using the same procedures as the bus manager, and appears to the driver as the bus manager itself. Thus, a memory request from a platform-independent device driver can be made in a secure manner.

Brief Summary Text (11):
According to one aspect, the present invention provides a computer-implemented method for allocating memory in which a bus manager is provided and configured to process memory allocation requests from a platform-independent device driver. The bus manager is an instantiation of an "outer" object class of bus managers. A platform-independent device driver is configured to generate memory allocation requests in terms of an abstract memory address space used by the bus manager. Such a request is generated and the identity if the device driver is determined. In addition, an "inner" class representation of the bus manager is generated. The inner class representation is configured to provide the same memory allocation request processing methods as the outer class bus manager. The request is processed and real memory is allocated to the device driver.

Drawing Description Text (10):
FIG. 9 is an illustration of a Java System Database entry in accordance with one embodiment of the invention.

Detailed Description Text (7):
Layers 202 include an applications layer 206. Layer 206 includes software applications that are commonly run by computer users. Such software includes word processors, graphics programs, database programs, and desktop publishing programs. These applications run in conjunction with runtime system 208. Runtime system 208 includes, in one embodiment, a Java Virtual Machine ("JVM") 210 that receives instructions in the form of machine-independent bytecodes produced by the application running in applications layer 206 and interprets the instructions by converting and executing them. Other types of virtual machines can be used with the present invention, however, as will be apparent to those of skill in the computer science and electronics arts.

Detailed Description Text (9):
Platform-dependent layers 204 include platform interface 224 that contains DMA classes 226, bus managers 228, and memory classes 230 in addition to other facilities that support runtime system 208. Additional functions are also included in platform interface 224 including interrupt classes (not shown) to support computer system 100. In one embodiment, these classes, managers, and functions are written in the Java programming language. Additional details about these features

can be found in the above-referenced, co-pending U.S. patent application Ser. No. 09/048,333.

Detailed Description Text (14):
In one embodiment, the above-described communication is performed using a hierarchical abstract memory object class and related sub-classes provided by the Java programming language. These class and sub-classes are described in detail in the above-referenced, co-pending U.S. patent application Ser. No. 09/048,333. It will be appreciated, however, that the methods, systems, and software described herein can be enabled using any similar programming language using techniques known to those of skill in the computer science arts.

Detailed Description Text (18):
One embodiment of a method by which the above-described drivers, managers, and objects function together to provide platform-independent drivers is described with respect to FIG. 5 at 500. At step 502 the device driver passes a request for memory allocation to the lowest-level bus manager. The bus manager creates an array of MemoryDescriptor objects that may be used by the device driver. The number of MemoryDescriptor objects in the array will depend on the number of types of memory object that the particular device driver may wish to allocate. That is, in the described embodiment, the list only contains MemoryDescriptor objects that the device driver may actually have a need to use. Although in theory, any number of MemoryDescriptor objects may be utilized by a particular device, in practice the number is typically quite small. Each of the entries in the MemoryDescriptor objects includes platform-dependent information in the AddressSpace and DevInfo object fields. The array may also be a "read-only" array, so that the device driver cannot modify the MemoryDescriptor objects. The array may be constructed in a variety of ways. In one embodiment, the JavaOS operating system is used and the bus manager constructs the array by first contacting the Java System Loader which creates a list of all associated MemoryDescriptor objects at start-up. In another embodiment, the array is constructed at the time the device driver is loaded. At step 504 the device driver uses the information contained in the DevInfo object field to identify the appropriate MemoryDescriptor object. In one embodiment, the device driver uses the information stored in the Base, Length, and AddressSpace object fields to select the appropriate MemoryDescriptor object. In another embodiment, the device driver uses the information stored in the Base, Length, AddressSpace, and DevInfo object fields to select the appropriate MemoryDescriptor object.

Detailed Description Text (26):
In some cases, it is desirable to implement the above-described allocation of device driver memory, or any other calls from a device driver to a bus manager (e.g., for other system services such as connecting, disconnecting, or interrogating interrupts) in a secure way. This is especially important where network-based drivers, such as drivers written in Sun Microsystems' Java programming language, are available from parties other than the operating system developer. For example, it may be desirable to prevent a "rogue" device driver from identifying itself to a bus manager as another, legitimate driver. This could be useful to improve system robustness, e.g., by allowing the system to identify and isolate corrupted device drivers and thus avoid system crashes. Such a security mechanism could also be useful to prevent "spoofing" device drivers from erasing, altering, and/or copying sensitive data from the system surreptitiously. By way of example, a malicious spoofing device driver could masquerade as a printer driver and copy the data to be printed to a remote location.

Detailed Description Text (28):
One embodiment of a method for providing the above-described secure bus manager support is shown in FIG. 8. The method described in this embodiment includes the use of procedures and facilities supported by JavaOS, but equivalent procedures and facilities for other operating systems will be apparent to those of skill in the

computer science arts. As shown at 800, in a first step 802, when the device driver is loaded by the Java System Loader ("JSL"), the device driver's object reference is passed to the bus manager assigned to service the device driver. The bus manager method getServingParent is called and passed the device driver's object reference. The device driver's object reference and bus manager assignment is determined by the JSL which "walks" down a "device tree" constructed by the Java System Database ("JSD") as illustrated in FIG. 9 at 900. There, an example tree includes an entry for Platform Manager 902 which has a "parent" relationship with Bus Manager 2904 and Bus Manager 1906. Bus Manager 2 has itself a parent relationship with the device drivers Device Driver 1908, . . . , Device Driver n 910. Bus Manager 1 has itself a parent relationship with the device drivers Device Driver n+1 912, . . . , Device Driver m 912. In this way, each registered device driver is assigned to an appropriate bus manager.

Detailed Description Text (30):
Returning once again to FIG. 8, at step 806 an object reference to the IC bus manager just constructed is passed to the device driver. The device driver, however, functions as if is was connected to the bus manager directly. Thus, by using inner class representations of the bus manager and making reference to the JSD for information on the device driver only those device drivers known to be associated with devices registered with the JSD can gain access to system resources. An additional level of security can be provided by forcing the device driver to choose the appropriate MemoryDescriptor object from a "read-only" list of such objects made available to the device driver by the bus manager.

Other Reference Publication (3):
Glenn Krasner, "The Smalltalk-80 Virtual Machine", Aug. 1981, BYTE Publications Inc., pp. 300-320 (even pages only).

CLAIMS:

1. A computer-implemented method for allocating memory in response to a memory allocation request from a platform-independent device driver, comprising the steps of: a. providing a bus manager, said bus manager being configured to process memory requests forwarded from a platform-independent device driver, and said bus manager being further configured to process such requests for memory allocation in terms of an abstract memory address space representation of said memory, said bus manager being an instantiation of an outer object class; b. providing a platform-independent device driver, said platform-independent device driver being configured to generate requests for allocation of memory in terms of said abstract memory address space representation used by said bus manager; c. generating a request for allocation of memory using said device driver and passing said request to said bus manager; d. verifying the identity of said device driver; e. generating an inner class representation of said bus manager in response to said request for memory, wherein said inner class representation of said bus manager is configured to provide at least the same memory request processing methods as said bus manager, said inner class representation of said bus manager; f. processing said request for memory allocation using said inner class representation of said bus manager to allocate memory for said device driver; and g. passing the real addresses of said allocated memory to said device driver.

4. The computer-implemented method of claim 3, further comprising the step of providing to said inner class representation of said bus manager said object reference for said device driver.

5. The computer-implemented method of claim 3, wherein said step of generating said inner class representation of said bus manager comprises creating an exclusive processing relationship with said device driver using said object reference for said device driver.

6. The computer-implemented method of claim 5, wherein said device driver is one a plurality of device drivers and said request for memory allocation is one of several requests generated by said plurality of device drivers and forwarded to said bus manager; and said method comprises the computer-implemented step of generating an inner class representation of said bus manager for each of said plurality of device drivers.

8. The apparatus of claim 7, further comprising a mechanism for generating an inner class representation of said bus manager.

10. The apparatus of claim 9, wherein said mechanism for verifying the identity of said device driver is configured-to-search said database for said association between said bus manager and said device driver in conjunction with generating said inner class representation of said bus manager.

11. The apparatus of claim 7, further comprising an inner class representation of said bus manager.

12. The apparatus of claim 11, wherein said inner class representation of said bus manager includes all memory request processing methods of said bus manager.

13. The apparatus of claim 12, wherein said inner class representation of said bus manager is associated uniquely with said device driver.

14. The apparatus of claim 13, further comprising a plurality of device drivers and inner class representations of said bus manager, wherein each of said plurality of said plurality of said inner class representations of said bus manager is uniquely associated with each of said plurality of device drivers.

15. A computer-readable medium containing computer-readable program code devices for allocating memory resources in a computer, said computer-readable program code devices being configured to cause a computer to execute the steps of: a. providing a bus manager configured to respond to requests for memory allocation from a device driver using an abstract address space representation of said computer memory; b. providing a device-independent device driver configured to request memory allocation in configured to generate requests for memory allocation in terms of said abstract address space representation of said memory; c. generating a request for memory allocation from said device driver to said bus manager; d. generating an inner class representation of said bus manager in response to said generated request; e. verifying the identity of said device driver; and f. processing said request using said inner class bus manager.

16. The computer-readable medium of claim 15, wherein said computer-readable program code devices are further configured to generate an inner class representation of said bus manager.

17. The computer-readable medium of claim 16, wherein said inner class representation is generated when said request is made to said bus-manager, and said inner class representation is assigned uniquely to said device driver.

19. A computer data signal on a carrier wave containing computer-executable instructions for allocating memory resources in a computer, said instructions being configured to cause the computer to execute the steps of: a. providing a bus manager configured to respond to requests for memory allocation from a device driver using an abstract address space representation of said computer memory; b. providing a device-independent device driver configured to request memory allocation in configured to generate requests for memory allocation in terms of said abstract address space representation of said memory; c. providing an inner class representation of said bus manager configured to process memory allocation requests from said device driver; d. generating a request for memory allocation

from said device driver to said bus manager; e. generating an inner class representation of said bus manager in response to said generated request; f. verifying the identity of said device driver; and g. processing said request using said inner class bus manager.

h e b b g e e e f c e f h e ge

First Hit Fwd Refs
End of Result Set

☐ Generate Collection Print

L31: Entry 1 of 1 File: USPT Nov 4, 2003

DOCUMENT-IDENTIFIER: US 6643711 B2
TITLE: Method and apparatus for dispatch table construction

Brief Summary Text (13):
Conventionally, every class and interface has a method table containing all the methods it locally defines. In addition, every class has a dispatch table, or Vtable, that has entries for all externally accessible methods that can be invoked by the class, including inherited methods. Typically private methods are note included in the Vtable. In conventional Vtable construction, every externally accessible method is associated with a single Vtable entry, which points to a section of code corresponding to a method. Thus, every Vtable entry points back at a method it corresponds to, whether local or inherited.

Detailed Description Text (5):
Additionally, each of the multiple entries for a particular method in a Vtable may be assigned a priority for the particular method in a class. Typically, the entry including the same accessibility designation as the accessibility of the method is assigned as the primary entry. The primary entry is typically invoked when the particular method is declared. In addition, a secondary and tertiary entry may also exist in the Vtable for the particular method corresponding to different accessibility designations.

Detailed Description Text (25):
When a method is called from an operating system 721, if it is determined that the method is to be invoked as an interpreted method, runtime system 719 can obtain the method from interpreter 717. If, on the other hand, it is determined that the method is to be invoked as a compiled method, runtime system 719 activates compiler 715. Compiler 715 then generates native machine instructions from bytecodes 705, and executes the machine-language instructions. In general, the machine-language instructions are discarded when virtual machine 711 terminates. The operation of virtual machines or, more particularly, Java.TM. virtual machines, is described in more detail in The Java.TM. Virtual Machine Specification by Tim Lindholm and Frank Yellin, 2.sup.nd Edition (ISBN 0-201-43294-3), which is incorporated herein by reference in its entirety.

Detailed Description Text (64):
In a particular embodiment, the above described Vtable building methods are implemented in the class representation of SUN microsystem's Hotspot's internal class representation. In this case, every class and interface has a method table containing all the methods it locally defines. In addition, every class has a dispatch table that has entries for all methods that can be invoked on an instance of the class (including inherited methods).

First Hit Fwd Refs

End of Result Set

☐ Generate Collection Print

L29: Entry 1 of 1 File: USPT Nov 4, 2003

DOCUMENT-IDENTIFIER: US 6643711 B2
TITLE: Method and apparatus for dispatch table construction

Detailed Description Text (62):
When a subclass implements a miranda method, the corresponding Vtable entry must be updated. Therefore, it must be separately determined if any miranda methods have been over-written. This is performed by searching through the superclasses' Vtable. More specifically, the Vtable is scanned from the bottom looking for miranda methods that match the name and descriptor of the current method. An entry 1712 corresponding to a miranda method is shown in the Vtable 1700.

L8: Entry 2 of 11 File: USPT Nov 4, 2003

DOCUMENT-IDENTIFIER: US 6643711 B2
TITLE: Method and apparatus for dispatch table construction

Brief Summary Text (10):
Accessibility is differentiated to facilitate a spectrum from unrestricted sharing of source code to specific forms of privacy. The two conventional types of accessibility levels are public and private. A public member can be seen or accessed by any other class. A private member can only be accessed by its own class. Particular languages may use additional accessibility types. For example, C++ uses a protected accessibility in which a protected member can generally only be accessed by its subclasses. Additionally, Java uses a package private accessibility in which a package private member can only be accessed by classes in a particular package. A package is a collection of related classes and interfaces that provide access protection and namespace management for all the elements of the package. Classes may be grouped into packages to make classes easier to find and use, to avoid naming conflicts, and to control access.

Brief Summary Text (11):
Generally, the accessibility of a message is specified upon declaration of the member. If no accessibility is initially provided, a default accessibility may be used. For example, within a Java package, the default accessibility for a member whose accessibility is not initially specified becomes package private.

Brief Summary Text (15):
The Java programming language contemplates the use of single inheritance. In Java, a subclass is defined to inherit all of the accessible members of its superclass and ancestors and can use these members as is, hide them or override them. Subclasses inherit those superclass members declared as public or protected. In addition, subclasses inherit a superclasses' members declared with no access designation as long as the subclass is in the same package as the superclass (i.e. they both default to package private). Thus, Java flexibly provides an accessibility spectrum from unrestricted sharing of source code to controllable levels of privacy. Conventional Vtable construction based on a diametric scheme for inheritance does not permit this accessibility spectrum.

Drawing Description Text (9):
FIG. 7a is a block/process diagram illustrating the transformation of a Java.TM. program containing Java source code to native code to be run on a particular platform or computer.

Drawing Description Text (10):
FIG. 7b is a diagrammatic representation of virtual machine, supported by the computer system of FIG. 18 described below.

Detailed Description Text (4):
In the described Java embodiment, four distinct accessibility designations are provided. Correspondingly, multiple searches up at least a portion of the superclass hierarchy may be performed for each designation. The search up the ascendant hierarchy ceases when all the accessibilities have been found in a superclass. As a class may also inherit from a superinterface, the semantic

interactions encountered between class hierarchies and accessibilities also occur in the interface hierarchy.

Detailed Description Text (19):
Any software language which uses accessibility constructs and inheritance of source code may implement the present invention. In a particular embodiment of the present invention, JAVA semantics in dispatch table construction implements the present invention.

Detailed Description Text (20):
FIG. 7a is a block diagram showing the inputs/outputs and the executing software/systems involved in creating native instructions from Java source code in accordance with one embodiment of the present invention. In other embodiments, the present invention can be implemented with a virtual machine for another language or with class files other than Java class files. Beginning with the left side of the diagram, the first input is Java source code 701 written in the Java.TM. programming language developed by Sun Microsystems of Mountain View, California. Java source code 701 is input to a bytecode compiler 703. Bytecode compiler 703 is essentially a program that compiles source code 701 into bytecodes. Bytecodes are contained in one or more Java class files 705. Java class file 705 is portable in that it can execute on any computer that has a Java virtual machine (JVM). Components of a virtual machine are shown in greater detail in FIG. 7B. Java class file 705 is input to a JVM 707. JVM 707 can be on any computer and thus need not be on the same computer that has bytecode compiler 703. JVM 707 can operate in one of several roles, such as an interpreter or a compiler. If it operates as a compiler, it can further operate as a "just in time" (JIT) compiler or as an adaptive compiler,. When acting as an interpreter, the JVM 707 interprets each bytecode instruction contained in Java class file 705.

Detailed Description Text (21):
FIG. 7b is a diagrammatic representation of virtual machine 711 such as JVM 707, that can be supported by computer system 1800 of FIG. 18 described below. As mentioned above, when a computer program, e.g., a program written in the Java.TM. programming language, is translated from source to bytecodes, source code 701 is provided to a bytecode compiler 703 within a compile-time environment 709. Bytecode compiler 703 translates source code 701 into bytecodes 705. In general, source code 701 is translated into bytecodes 705 after the time source code 701 are created by a software developer.

Detailed Description Text (22):
Bytecodes 705 can generally be reproduced, downloaded, or otherwise distributed through a network, e.g., through network interface 1824 of FIG. 18, or stored on a storage device such as primary storage 1804 of FIG. 18. In the described embodiment, bytecodes 703 are platform independent. That is, bytecodes 703 may be executed on substantially any computer system that is running a suitable virtual machine 711. Native instructions formed by compiling bytecodes may be retained for later use by the JVM. In this way the costs of the translation are amortized over multiple executions to provide a speed advantage for native code over interpreted code. By way of example, in a Java.TM. environment, bytecodes 705 can be executed on a computer system that is running a JVM.

Detailed Description Text (23):
Bytecodes 705 are provided to a runtime environment 713 which includes virtual machine 711. Runtime environment 713 can generally be executed using a processor such as CPU 1002 of FIG. 18. Virtual machine 711 includes a compiler 715, an interpreter 717, and a runtime system 719. Bytecodes 705 cart generally be provided either to compiler 715 or interpreter 717.

Detailed Description Text (25):
When a method is called from an operating system 721, if it is determined that the

method is to be invoked as an interpreted method, runtime system 719 can obtain the method from interpreter 717. If, on the other hand, it is determined that the method is to be invoked as a compiled method, runtime system 719 activates compiler 715. Compiler 715 then generates native machine instructions from bytecodes 705, and executes the machine-language instructions. In general, the machine-language instructions are discarded when virtual machine 711 terminates. The operation of virtual machines or, more particularly, Java.TM. virtual machines, is described in more detail in The Java.TM. Virtual Machine Specification by Tim Lindholm and Frank Yellin, 2.sup.nd Edition (ISBN 0-201-43294-3), which is incorporated herein by reference in its entirety.

Detailed Description Text (64):
In a particular embodiment, the above described Vtable building methods are implemented in the class representation of SUN microsystem's Hotspot's internal class representation. In this case, every class and interface has a method table containing all the methods it locally defines. In addition, every class has a dispatch table that has entries for all methods that can be invoked on an instance of the class (including inherited methods).

Detailed Description Text (67):
FIG. 18 is a block diagram of a general purpose computer system 1800 suitable for carrying out the processing in accordance with one embodiment of the present invention. For example, JVM 707, virtual machine 711, or bytecode compiler 703 can run on general purpose computer system 1800. FIG. 18 illustrates one embodiment of a general purpose computer system. Other computer system architectures and configurations can be used for carrying out the processing of the present invention. Computer system 1800, made up of various subsystems described below, includes at least one microprocessor subsystem (also referred to as a central processing unit, or CPU) 1802. That is, CPU 1802 can be implemented by a single-chip processor or by multiple processors. CPU 1802 is a general purpose digital processor which controls the operation of the computer system 1800. Using instructions retrieved from memory, the CPU 1802 controls the reception and manipulation of input information, and the output and display of information on output devices.

CLAIMS:

10. The dispatch table of claim 9 wherein the object oriented language is JAVA.

19. The dispatch table of claim 17 wherein the object oriented language is JAVA.